Chorus:A communication and processing architecture for distributed systems

H. Zimmermann, M. Guillemont, G. Morisset, J.S. Banino

IRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 328

CHORUS: A COMMUNICATION AND PROCESSING ARCHITECTURE FOR DISTRIBUTED SYSTEMS

Hubert ZIMMERMANN
Marc GUILLEMONT
Gérard MORISSET
Jean-Serge BANINO

Septembre 1984

CHORUS:
A communication and processing architecture for distributed systems

Hubert ZIMMERMANN *

Marc GUILLEMONT **

Gérard MORISSET **

Jean-Serge BANINO **

* CNET
38, 40 rue du Général Leclerc
92131 Issy les Moulineaux
FRANCE

** INRIA
BP 105
78153 Le Chesnay Cedex
FRANCE

SUMMARY

CHORUS is an architecture for distributed systems, designed for a large class of machines and applications. The CHORUS architecture is built with a small set of powerful concepts. It is a solid and sound basis for building distributed applications, for learning distribution as well as for new researches and experiments.

CHORUS brings a new apprach to distribution where communications play a key-role : expressed in the design of a distributed application, they drive the process execution and induce synchronization. More, they render distribution transparent at run-time : all communications are uniform whatever be the entities which communicate and whatever be their respective locations.

That communication is acheived by the means of messages exchanged through ports which constitute the logical communication interface. Communication is integrated in the nucleus of the underlying operating system.

The paper presents in detail the CHORUS architecture and its operating system. Some aspects of CHORUS are then analyzed more deeply and discussed. Finally, a simple example and two implementations illustrate various aspects of the architecture.

RESUME

CHORUS est une architecture de système réparti conçue pour une grande variété de machines et d'applications. Bâtie autour d'un petit ensemble de concepts simples et puissants, CHORUS est une base solide et saine pour le développement d'applications réparties, l'apprentissage de la répartition et la recherche.

CHORUS propose une approche originale de la répartition où les communications jouent un rôle fondamental : exprimées dès la conception de l'application, elles structurent l'exécution des processus et en déterminent la synchronisation. De plus, elles rendent la répartition transparente à l'exécution : toutes les communications sont uniformes quelles que soient les entités qui communiquent et quelles que soient leurs localisations respectives.

Cette communication est réalisée au moyen de messages échangés entre des portes qui constituent une interface logique de communication. Elle est intégrée au coeur du système d'exploitation sous-jacent à l'architecture.

Cet article présente en détail l'architecture CHORUS et son système exécutif. Il analyse et justifie les choix effectués. Un exemple simple et deux implantations illustrent cette architecture.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

1/ Introduction

CHORUS ([Banino 80], [Zimmermann 81], [Guillemont 82a]) is an architecture for distributed systems. It has been designed for a wide variety of machines and networks and for a large class of applications : process control, mail systems, office automation, telecommunication, etc... It includes a basic methodology for designing distributed applications, a structure for executing them and the (operating) system supporting this execution.

The CHORUS project was launched at INRIA in 1979. This paper is an overview of CHORUS, five years after its initiation : results and perspectives. The first two sections present the CHORUS architecture and the underlying system; discussion and justifications of CHORUS' choices may be found in section 4; section 5 illustrates with an example the various aspects of CHORUS; finally, section 6 deals with two implementations of CHORUS.

2/ The CHORUS distributed architecture

In CHORUS, a distributed system spreads over a set of interconnected sites (local systems). On each site, processing is performed by active entities called actors; the concept of actor is an adaptation of the traditional concept of sequential process for the aim of distributed and secure systems. An actor may create and destroy other actors on its own site and on distant sites as well; it may communicate with other (local or distant) actors by means of messages exchanged through ports. Actors, ports and messages may be created and detroyed dynamically.

On each site the CHORUS operating system supports and controls the operation of actors and provides actors with system *services*. On each site the CHORUS system includes a local *kernel* and a set of system actors.

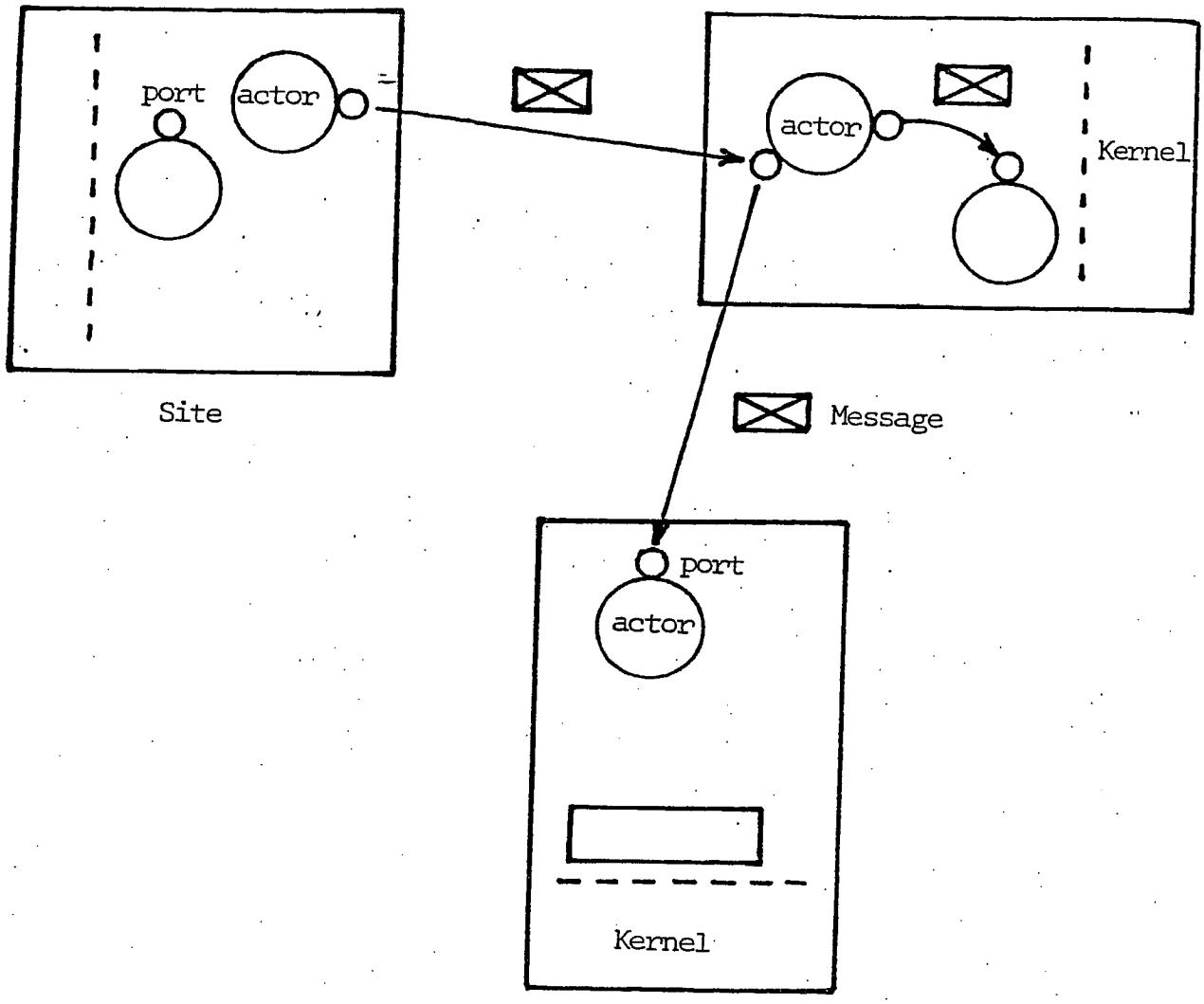


figure 2.1 : Overview of the CHORUS architecture

Section 2.1 introduces the background choices in the definition of the CHORUS architecture, while a more detailed presentation of this architecture can be found in sections 2.2 through 2.6.

2.1/ Foundations

The definition of the CHORUS architecture derives largely from four "foundation-stones" introduced hereafter and respectively related to :

(a) Structuring execution of distributed applications

(b) Structuring execution of actors

(c) Unifying communications

(d) Standardizing access protocols

2.1.1/ Structuring_execution_of_distributed_applications

Distributed processing has often been modeled by communicating processes i.e. by traditional processes performing local I/O operations in order to send and receive the messages they wish to exchange. This process view of distribution through local operations makes it often difficult to describe protocols, since these are primarily concerned with the exchange of messages (i.e. distributed operations), rather than with individual local send and receive operations.

CHORUS takes the view that processing and communication play symmetrical, though complementary, roles : the operation of a distributed application consists of communication_steps (message transfer) and processing_steps (message processing) which trigger each other (see figure 2.2). One may thus consider that processing drives communication (traditional data-processing view) or that communication drives processing (traditional protocols view). Depending on the problem to solve, one view or the other or a combination of both will be best adapted to organize and

describe distributed activities.

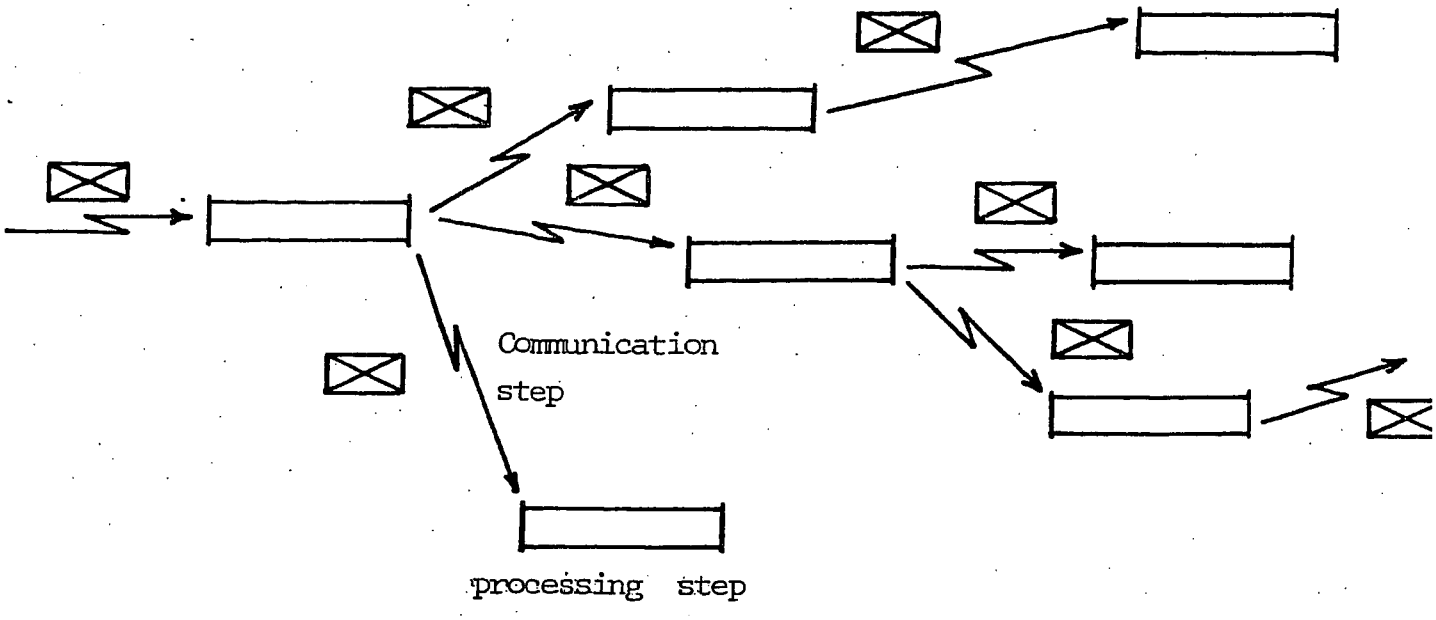


figure 2.2 : Execution of a distributed application

2.1.2/ Structuring execution of actors

A **processing-step** is the elementary quantum of processing performed by an **actor** : it is triggered by the receipt of one message and results in the transmission of n other messages (n>=0).

An actor is the most elementary processing entity considered in CHORUS. It is purely local, i.e. entirely - code and data - on one site. It is purely sequential, i.e. within an actor, as illustred in figure 2.3, there is no interweaving of processing-steps; messages received are processed one by one; each message received triggers one processing-step and only the receipt of a message may trigger a processing-step.

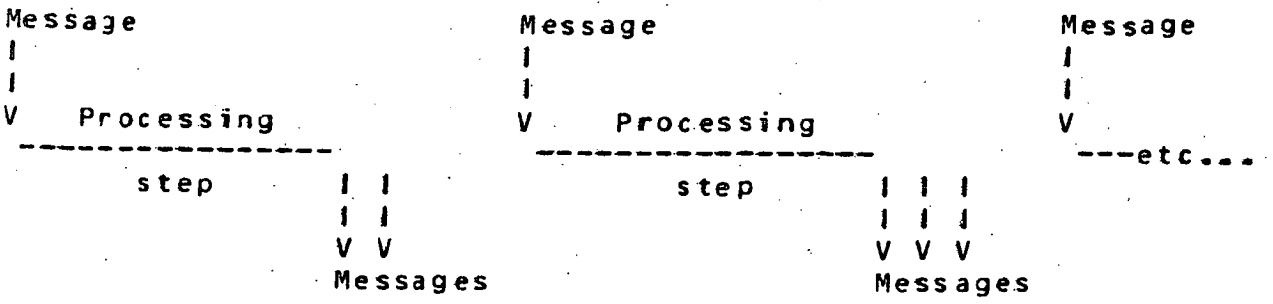


figure 2.3 : Operation of an actor

2.1.3/ Unifying communications

All communications among actors take the form of an exchange of messages. A message is sent from one port of the sending actor - the source port - onto one (or several) port(s) of the receiving actor(s) - the destination port(s). Communications between an actor and the system also take the form of exchanges of messages between ports. The basic communication service provided by the CHORUS system is of the connectionless type [ISO 82]; it is location independent, i.e. local and remote communications look the same to actors exchanging messages. In the following, a message will be represented by the triplet

Source port, Destination port(s), [Text of the message]

2.1.4/ Standardizing access protocols

In order to access to a service a user actor must communicate with the provider of that service. In CHORUS, this communication takes the form of an exchange of messages between user and provider according to a service access protocol.

The most common type of service access protocol is the "Request-Response" type, illustrated in figure 2.4, in which

- the service request is sent in a message from port Pa of the user requesting the service onto port Ps representing the provider of the service,
- the service response is sent back in a message from port Ps onto port Pa.

Request : Pa, Ps, [Request parameters]

Response : Ps, Pa, [Response parameters]

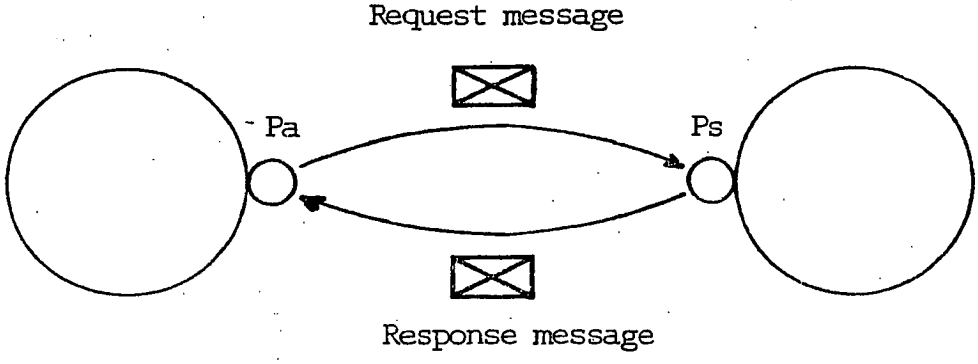


figure 2.4 : Request-Response exchange with a service

Another type of basic service access protocol is the "Order" type, illustrated in figure 2.5, in which

- the service order is simply sent from port Pa of the user ordering the service onto port Ps representing the provider of

the service,

- no answer needs to come back from the server.

Order : Pa, Ps, [Order parameters]

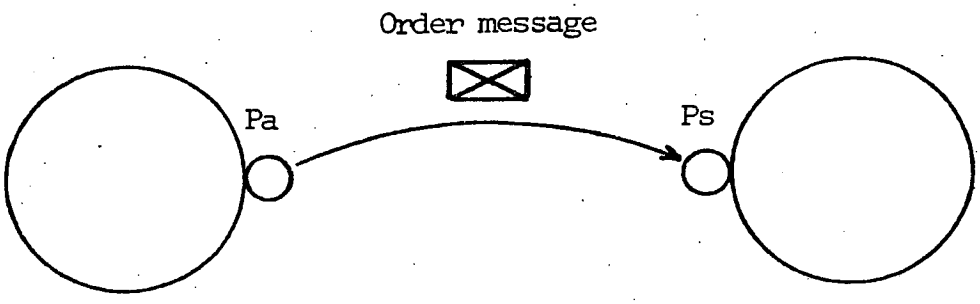


figure 2.5 : Order to a service

Both the Request-Response and the Order type of service access protocols are extensively used in CHORUS for access to system services.

2.2/ Internal operation of an actor

Each actor performs a sequence of message processing-steps. Each processing-step is preceeded by two operations :

(a) a **selection** (performed by the system) which determines which message is to be processed next by the actor, and

(b) a **switch** (performed by the system) which determines which piece of code in the actor is to be executed in this processing-step.

These three elementary phases of operation of actors (selection, switch, processing-step) are described in more detail in the following subsections 2.2.1 to 2.2.3.

2.2.1/ Selection_service

Messages received by an actor are queued at their respective destination ports. At a given instant, several messages may be queued at one or several ports of an actor. A selection (performed by the system) makes it possible to decide, at the end of each processing-step, which message is to be processed by the actor during the next processing-step.

This selection may be dynamically parameterized by the actor calling upon a system service by sending to it the order message

Order_to_the_service_:

Port, Select, [list of (Sending port, Receiving port)]

- Port is the name of a port of the actor used to send the order.
- Select designates the port representing the selection service.
- Sending port is the name of a port.
- Receiving port is the name of a port of the actor on which the selection applies.

Effect : messages received onto one of the "Receiving port" in the list and coming from the corresponding "Sending port" are candidate for selection, i.e. for being processed by the actor at the next processing-step : each couple (Sending port, Receiving port) constitutes a kind of filter

to select the next message to be processed, and all these filters operate in parallel.

Conventions : the couple (All, Receiving port) in the list means that the actor is ready to accept any message received onto "Receiving port" while the couple (All, All) means acceptance of any message received onto any of its ports.

Note : this service has an access protocol of the Order type, i.e. does not return an answer. It simply records parameters for the selection. If there is an error in the order message, the wrong couple is not registred; if all couples are wrong, the selection list is taken as (All, All).

If several messages fulfill the selection conditions (i.e. pass each through one of the filters), the selection service takes into account the priorities of ports (see section 2.3.4) and selects the message received onto the port with the highest priority. Finally, if several messages fulfill these two conditions, the selection service selects the message which was received first.

The selection conditions remain unchanged until a new selection order is passed to the system. For the sake of simplicity, each selection order redefines the whole of the selection conditions (there is no order for partial modification). Finally, the initial value of the selection conditions is (All, All).

2.2.2/ Switch_service

The code to be executed in a processing-step is designated by an entry-point in the actor's code; each actor can define several entry-points which correspond to the various processing-steps it may execute. A switch (performed by the system) makes it possible to determine which entry-point is to be entered to process the message which has just been selected by the selection service.

This switch may be dynamically parameterized by the actor calling upon a system service by sending to it the order message

Order_to_the_service_:
Port, Switch, [Entry-point]

- Port is the name of a port of the actor to which the switch order applies.
- Switch designates the port representing the switch service.
- Entry-point designates an entry-point in the actor's code.

Effect : a message received onto "Port" and subsequently selected by the selection procedure will trigger the execution of a processing-step starting at "Entry-point".

Each such switch order modifies only the designation of the entry-point associated with "Port", while the designation of the entry-points associated with the other ports of the actor (if any) remain unchanged.

The initial value of the switch parameters is the association (Umbilical port, Initial entry-point) (see section 2.4).

2.2.3/ Processing-step

When a message has been selected and switched, the processing-step is started in the actor. During the execution of a processing-step, the actor has access to the message which triggered the processing-step and only to this message; it has also access to its internal data. The processing-step ends with the execution of a RETURN primitive which triggers the transmission of all messages prepared by the actor during the processing-step.

The programming of a processing-step may therefore be represented as follows :

```
ENTRY-POINT e;
.....
{processing of the message}
.....
RETURN (Ps1, Pd1, [Text 1];
        .....
        Psn, Pdn, [Text n]);
```

figure 2.6 : Programming a processing-step

The "Psi, Pdi, [Text i]" are the messages sent when the processing-step ends; Psi are ports of the actor.

Note : all examples in this paper will use this schematic representation of processing-steps which is intended only to be illustrative. The question of a programming language for actors is discussed in section 4.6.

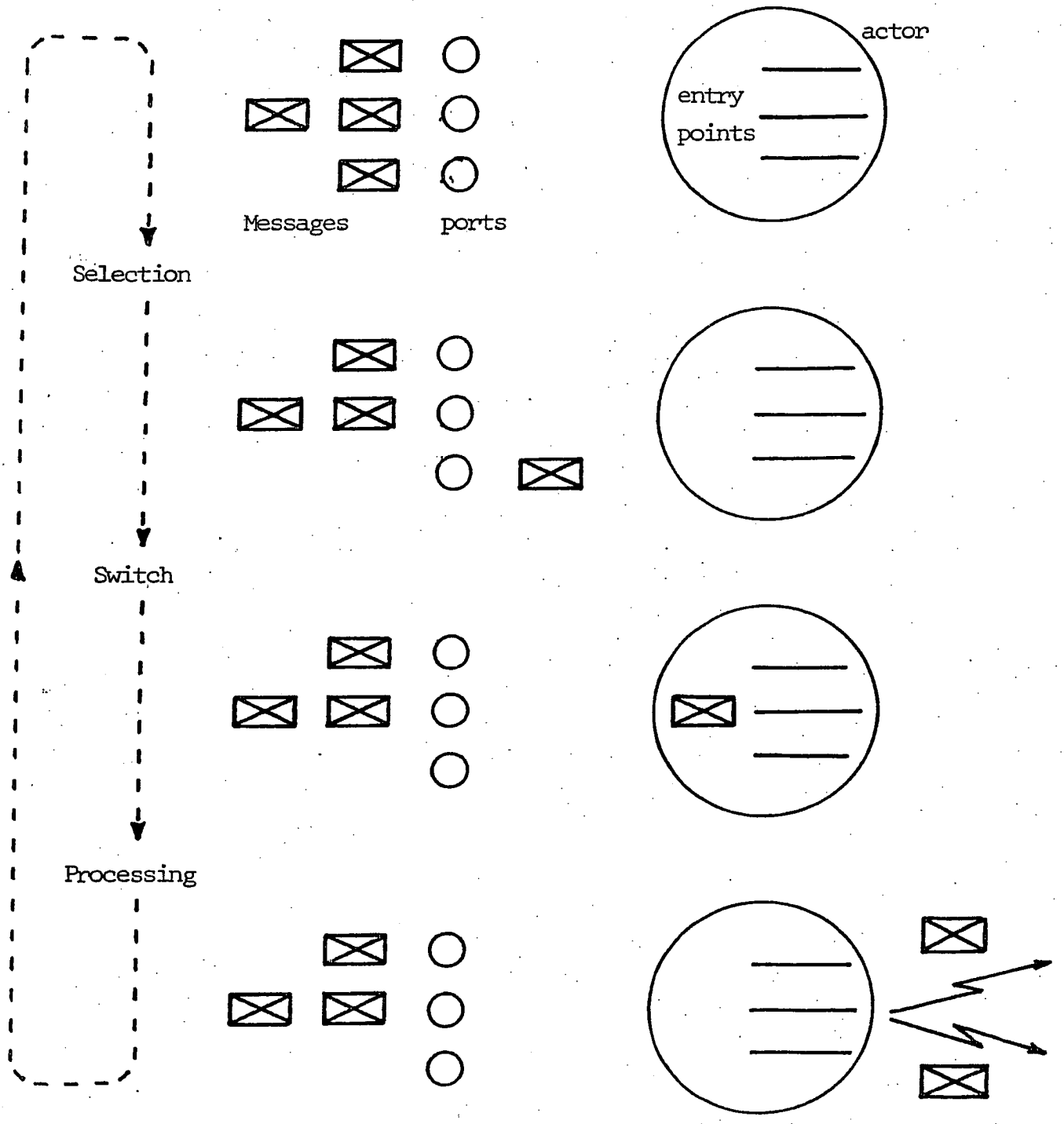


figure 2.7 : Operation of an actor

2.3/ Communication_services

An actor may communicate with its outside world (i.e. other actors and the system) by sending and receiving messages through ports. The corresponding message transfer service and the associated time-out service are described in sections 2.3.1 and 2.3.2.

In order to be used for sending or receiving messages, a port must first exist, and second be open and linked to an actor. System services are available for the management of ports. Port creation and port destruction services are described in section 2.3.3, while port opening and closing services are described in section 2.3.4.

2.3.1/ The_message_transfer_service

The message transfer service allows actors to send and receive messages through ports. The same port may be used both for sending and receiving messages : ports are bidirectionnal. Messages are transfered from one source port to one (or more) destination ports(s) along with the indication of the names of both source and destination ports.

The message transfer service makes distribution transparent to actors in the sense that they need not know where (i.e. on which site) their correspondents are located; they only need to know the names of their correspondents' ports and refer to these when sending messages; the message transfer service itself will find the site where each destination port resides and transfer the message onto that site (see section 3.6).

This basic communication facility is a real-time connectionless facility. No connection needs to be established prior to sending or receiving messages and successive message transfers are considered (by the service) as independent operations. Messages are transfered as fast as possible; if a message cannot be delivered within a given delay, the service will drop it. Finally, there is no report back from the service to indicate the result, success or failure, of the message transfer operation.

Four points are worth being noted about this real-time connectionless service :

(a) The real-time, non blocking properties of this service are essential in real-time applications.

(b) More sophisticated types of communication facilities such as the OSI transport connections [ISO 82] or the satellite bulk transfer facility of NADIR [Grangé 82] and many others, can be built on top of the CHORUS basic message transfer service.

(c) Local message transfer, if properly handled, is as reliable as traditional program-to-program communications (through procedure calls, system calls or queues).

(d) CHORUS offers a built-in time-out service which is the basis for detecting the non-arrival of messages (see section 2.3.2).

Contrary to all other services, and for obvious reasons, the message transfer service is not invoked by sending a message to it. As mentionned in section 2.2.3, it is invoked implicitely at the end of each processing-step when the RETURN primitive is executed. At that time, all messages mentionned in the RETURN primitive are taken into account by the message transfer service.

Each message contains the following informations :

Source port, Destination port(s), [Text of the message]

- Source port is the name of a port (of the sending actor) through which the message is sent.
- Destination port is the name of a port towards which the message should be transfered.

The validity of messages to be sent and received through a given port can be checked by send and receive control procedures (see section 2.5).

The internal functioning of the message transfer service is discussed further in section 3.6.

2.3.2/ The time-out service

Since a processing-step may be triggered only by the receipt of a message, an actor might be blocked, waiting for a message which will never come (for instance because the expected sender of the message has failed or because the transmission of the message was not possible). In order to avoid such possible deadlocks, an actor can enable a time-out on a "liaison" (i.e. a couple "Sending port", "Receiving port") by sending an order message to the time-out service

Order_to_the_service_:

Port, Time_Out, [Sending port, Receiving port, Delay]

- Port is the name of a port of the actor.
- Time_Out designates the port representing the time-out service.
- Sending port is the name of a port.
- Receiving port is the name of a port of the actor.
- Delay is a delay.

Effect :

- if no message sent by "Sending port" has been received onto "Receiving port" when "Delay" has elapsed, the system generates the diagnosis message (according to the Request-Reply protocol)

Time_Out, Port, [Sending port, Receiving port]

This diagnosis message, as any message, will trigger a processing-step in the actor. This message is "filtered" by the selection service as a message sent by "Sending port" and received onto "Receiving port".

- if a message sent by "Sending port" is received onto "Receiving port" before "Delay" has elapsed, the time-out is disabled.

Conventions : the convention "Port, Time_Out, [All, Receiving Port, Delay]" allows an actor to

enable a time-out which is disabled by any message received onto "Receiving port".

Note : several time-outs with different "Sending port" may be enabled simultaneously on the same port.

In other words, this mechanism watches over the "liaison" ("Sending port", "Receiving port") and, in case of error, a diagnosis message is received onto "Port".

2.3.3/ Port_creation_and_destruction_services

Ports are created from models_of_port which contain in particular the send and receive control procedures to be associated with the port (see section 2.5). Several ports may be created from the same model of port. At the time of its creation, a port is given a unique global name which will be used subsequently to refer to it.

An actor may request the creation of a port calling upon the port creation service by sending to it the request message

Request_to_the_service_:

Port, Create_Port, [Model, Name, Initial parameters]

- Port is a port of the actor.
- Create_Port designates the port representing the port creation service.
- Model is the name of the model of port which describes the port to be created.
- Name is the unique global name of the port to be created; if this parameter is not present, the port is given a new unique global name choosen by the system.
- Initial parameters are used for initializing the control procedures associated with the port.

Response_from_the_service_:

Create_Port, Port, [Diagnosis, Name of the created port]

Conversely, the destruction of a port may be requested by calling upon the port destruction service by sending to it the request message

Request_to_the_service_:

Port, Destroy_Port, [Name]

- Name is the name of the port to be destroyed.

Response_from_the_service_:

Destroy_Port, Port, [Diagnosis]

Note : in order to be destroyed, a port must be closed (see section 2.3.4); otherwise, the destruction is rejected.

Creation and destruction of ports are controlled by port-creation and port-destruction control procedures (see section 2.5) associated with the port.

2.3.4/ Port opening and closing services

Before using a port (either to receive or to send messages or both), an actor must open that port; the open operation is similar to linking the port to the actor. When the actor does not need to use the port any longer, it closes it (i.e. it "unlinks" it). A port may be linked to only one actor at a time, but the same port may successively be linked to different actors.

Messages sent onto a port which is not opened (i.e. which is not "linked" to any actor) are lost.

An actor may request the opening of a port calling upon the port opening service by sending to it the request message

Request to the service :

Port, Open_Port, [Name, Priority]

- Name is the name of the port to be opened.
- Priority is the priority allocated to this port and used for the scheduling of actors (see section 3.1.1).

Response from the service :

Open_Port, Port, [Diagnosis]

The opening of a port is controlled by an open control procedure (see section 2.5). This control may be used, for instance, to prevent an actor from opening a port representing a service it is not entitled to offer.

Conversely, an actor may request a port to be closed, calling upon the port closing service by sending to it the request message

Request_to_the_service_:
Port, Close_Port, [Name]

- Name is the name of the port to be closed, which must be currently linked to the actor issuing the request.

Response_from_the_service_:
Close_Port, Port, [Diagnosis]

Note_: an actor may close only the ports it has succesfully opened (i.e. the ports which are "linked" to it).

There is no close control procedure since any actor may close any of the ports attached to it (i.e. it has opened and not yet closed) except for its "umbilical" port (see section 2.4).

2.4/ Actors_creation_and_destruction_services

An actor is created from a *model_of_actor* which contains in particular its code and data (see section 2.6). Several actors may be created from the same model of actor.

An actor is created with an "*umbilical*" port on which it receives its *initial_message*; the umbilical port is automatically switched onto an *initial_entry-point*; therefore, the initial message triggers the execution of the intial step.

An actor may request the creation of other actors, locally or on a distant site by calling upon the actor creation service. An actor may request its own termination or the destruction of other actors by calling upon the actor destruction service. Request and Response messages exchanged for the creation and destruction of actors are analogous to those used for accessing port creation and destruction services (see section 2.3.3).

Similarly, creation and destruction of actors are controlled by actor-creation and actor-destruction control procedures (see section 2.5).

2.5/ Protection

In CHORUS, each actor with its ports is considered to be an elementary protection domain. As suggested in sections 2.3 and 2.4, where various control procedures have been mentionned, CHORUS permits to control formation of domains as well as interactions among domains; conversely, CHORUS does not control the internal behaviour of actors during processing-steps.

The protection mechanisms of CHORUS can be divided into two classes :

- built-in controls which enforce architectural rules and are automatically performed by the system (such as "a port can be attached to only one actor at a time" or "an actor may send a message only through a port it has opened", etc...).
- control procedures which enforce user defined rules, and thus permit to adapt protection to each application or class of applications.

Control procedures available in CHORUS are summarized in the table of figure 2.8 :

System service	Control procedure	Associated with
Actor creation	yes	model of actor
Actor destruction	yes	actor
Port creation	yes	model of port
Port destruction	yes	port
Port opening	yes	port
Port closing	no	
Message transmission	yes	port
Message receipt	yes	port

figure 2.8 : Control procedures available in CHORUS

Control procedures are executed by the system on each site upon receipt of the corresponding request.

For instance, if actor A requests the destruction of actor B, the system executes the "actor destruction control procedure" associated with B; if the result is positive, the destruction request is considered valid and executed; if the result is negative, the destruction request is considered invalid and not executed. In every case, actor A receives a diagnosis corresponding to the result of its request in the response message coming back from the service.

Another important usage of control procedures is the control of a message transfer; e.g. message "Pa, Pb, [M]" is transfered from port Pa opened by actor A to port Pb opened by actor B.

- when the message is transmitted through Pa, the system executes the "send control procedure" associated with Pa; this

procedure checks for instance that M is consistent with the specifications of actor A; if the control is negative, the message is destroyed and the system may generate a diagnosis message (see section 3.4).

- when the message arrives at port Pb, the system executes the "receive control procedure" associated with Pb; this procedure checks for instance that actor B is permitted to process M; if the control is negative, the message is destroyed and the system may generate a diagnosis message (see section 3.4).

These control procedures are user defined and may therefore be closely adapted to each application; they may also for instance be changed (without any change in the actors code) when moving from a debugging phase to real operation. In the future, as experience grows, standard control procedures will be made available to users within system's libraries.

2.6/ Construction of an actor

As already stated in section 2.4, actors are created from models of actors. All actors created from the same model have the same code and identical initial data segments. Initial parameters specific to each actor are passed in its initial message.

CHORUS permits to define models of actors in two steps :

(a) A distributed application (or part of it) is described as a set of **modules** communicating through ports, just as actors would do.

(b) The distributed application (or part of it) is then configured by grouping modules into models of actors, each model being

obtained from one or several modules in addition with control procedures (see section 2.5).

The grouping of several modules into one model of actor makes it possible to reduce the associated overhead (e.g. communication between modules in the same model of actor can be expedited by replacing a message transfer by a procedure call). However, this grouping imposes that all these modules be executed on the same machine and belong to the same protection domain.

Of course, a distributed application can be reconfigured (e.g. for reducing overhead, or for distributing work load, or etc...) without changing its description in terms of modules.

3/ The CHORUS system

As already outlined, the CHORUS system which supports the execution of the CHORUS architecture consists of a kernel (resident on each site) and system actors (which need not all be present on each site). This section presents the organization of the CHORUS system including the management of interrupts and I/Os which have been integrated within the CHORUS architecture.

3.1/ Structure of the system

3.1.1/ The kernel

On each site, the kernel provides basic support for the operation of local actors by performing the following functions :

- management of selection conditions and scheduling of actors,
- management of switch conditions,
- management of time-outs,
- local transfer of messages.

In addition, the kernel participates in

- management of interrupts (see section 3.2),
- realization of I/Os (see section 3.3).

The operation of the kernel can be viewed as a cyclic sequence of three phases : selection, switch and return as explained below. Note that, due to interrupts and priorities and to multiprocessing, the kernel may manage several such cycles in parallel.

Selection_phase

The kernel selects, as the next message to be processed, a message M queued at port P of actor A, according to the following criteria :

(a) Actor A must not be currently executing a processing-step (as there must be no interweaving of processing-steps within an actor),

(b) Message M must fulfill the selection conditions defined by actor A for port P (see section 2.2.1),

(c) If several ports remain after applying criteria (a) and (b) above, select as P one of the ports with the highest priority (see section 2.3.4),

(d) If several messages remain selectable at port P after applying criteria (a), (b) and (c) above, select as M the message which was received first.

Switch_phase

When message M has been selected, actor A becomes active. The kernel then switches message M onto the right entry-point in A (according to switch conditions given by A for port P, see section 2.2.2), installs the proper context and triggers the execution of the processing-step.

Return_phase

When the processing-step ends (with a RETURN primitive), the kernel examines all messages transmitted by the actor and :

(a) processes the messages addressed to one of its own ports (these messages correspond to requests of services performed by the kernel, see section 3.1.4),

(b) transfers other messages to local ports (see section 3.6).

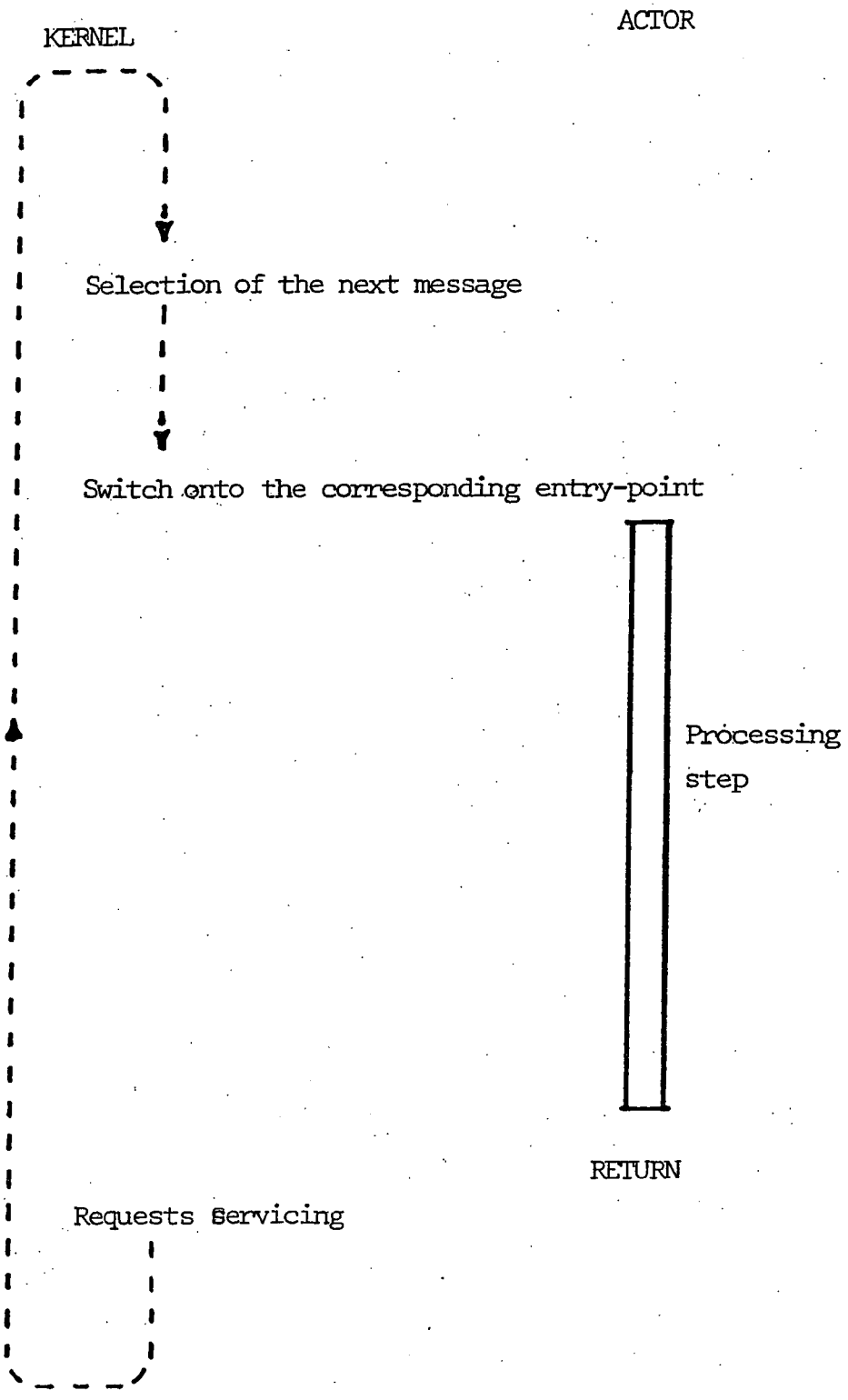


figure 3.1 : Schematic operation of the kernel

3.1.2/ System_actors

System services which are not performed by the kernel are supported by system actors, in particular :

- creation and destruction of actors,
- creation, destruction, opening and closing of ports,
- remote communication (see section 3.6),
- memory management,
- management of names,
- console management,
- file management,
- etc...

System actors differ from application actors only by the fact that they have specific rigths and may have privileged communications (by messages) with the kernel, e.g. to indicate modifications of local actors or ports (see section 3.1.3) or to request I/Os (see section 3.3).

3.1.3/ Relation_kernel_/_system_actors

In order to perform its functions, the kernel needs informations like names of local actors and ports, associations between ports and actors, etc... These informations are stored in two tables :

(a) the table of local actors which contains for each actor : the name of its model, its name, its status, its context of execution, reference of its ports, parameters for selection.

(b) the table of local opened ports (and those ports only) which contains for each port : its name, priority, reference of the

actor which has opened it, queue of messages received, time-outs (if any), parameters for switch.

When a system actor processes a service request, it may have to update the tables of the kernel; for instance, if an actor requests the opening of port P, the port P must be added in the table of local opened ports. For the sake of modularity and protection, system actors do not access directly the tables of the kernel; they rather transmit to the kernel a request message like "port P is opened" causing the kernel to update its tables accordingly. Therefore, system actors do not need to know the structure of the kernel tables.

This clear separation between the kernel and system actors has several advantages :

- the interface to the kernel is clearly specified by the list of messages it can receive;
- the format of the tables in the kernel may change without any change in system actors.
- the code of system actors is more machine independent and is therefore easier to transport.

3.1.4/ System_interface

For the sake of uniformity, the system appears to actors as a set of ports : the kernel as well as system actors is viewed as receiving and sending messages through ports. The access to any service follows one of the service access protocols described in section 2.1.4 (Request-Response or Order) whichever entity performs this service. This offers several advantages :

- an actor does not need to know the distribution of services between the kernel and system actors.
- this distribution of services between the kernel and system actors may be changed without any change in application actors.
- some system actors may be remote without this being visible to user actors since the communication service is site independent. (The only constraint is that system actors which transmit information to the kernel must be on the same site as the kernel.)

For instance, in the current implementation of CHORUS, the ports of the kernel are the following : Select, Switch, Time_Out (see sections 2.2.1, 2.2.2 and 2.3.2) and some others for communication with system actors.

Other ports, like Create_Port, Destroy_Port, Open_Port, etc... (see section 2.3) are ports of system actors.

On each site, the minimum system consists of a kernel and a small set of system actors :

- an "actor manager",
- a "port manager",
- a "transport station",

- a "console manager" on sites which have a console,
- a "file manager".

Other system actors may be remote.

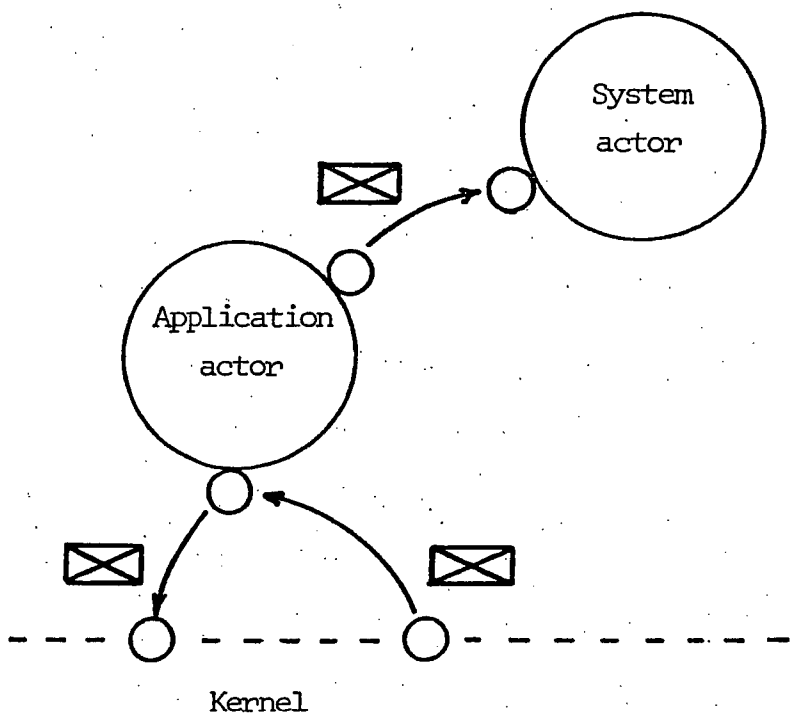


figure 3.2 : System interface

3.2/ Interrupts

Interrupts do not activate actors directly : this would not fulfill within the general scheme of operation of an actor where only a message may trigger a processing-step. Interrupts are seen by actors as interrupt-messages sent by interrupt-ports according to the following scheme :

A system actor IMA (Interrupt Manager Actor) opens one port Pi for each interrupt-level i; the port Pi represents the interrupt mechanism i for other actors. When an actor A needs to

receive interrupt-messages corresponding to interrupts of level i, it sends a request-message onto Pi :

Pa, Pi, [association with the interrupt-level]

The actor IMA receives this message and checks the validity of the request : if the control is positive, IMA transmits to the kernel the triplet (i, Pi, Pa) :

Ps, Pk, [i, Pi, Pa]

The kernel keeps this triplet in a table and enables interrupt-level i.

When interrupt i arises, the kernel receives it and generates an interrupt-message

Pi, Pa, [an interrupt has arised]

This message will trigger a processing-step, exactly as any other message.

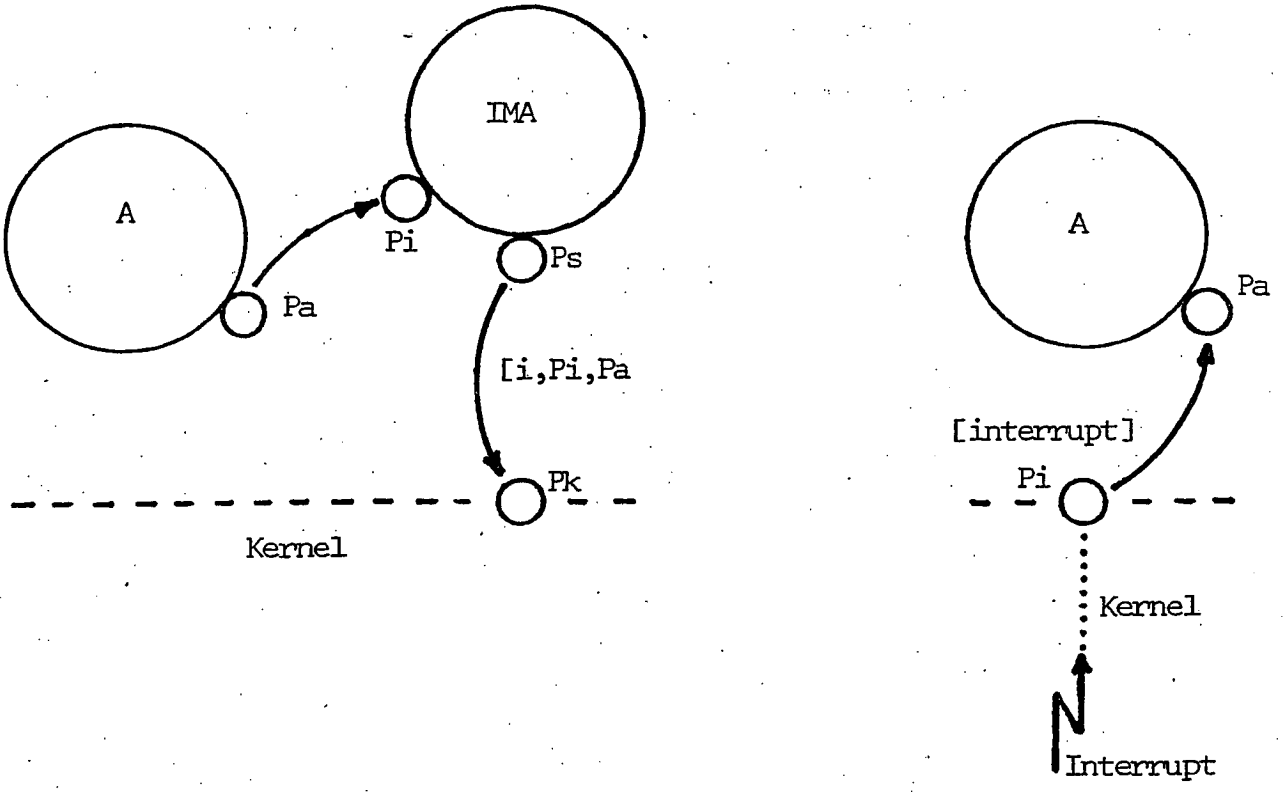


figure 3.3 : Implementation of interrupts

Depending on requirements for each site, preemption may be authorized or not. If authorized, when the kernel has transformed an interrupt into a message onto port Pa, the current active actor is suspended and the processing-step associated with port Pa is started. Preemption allows faster response time for high priority interrupts and is therefore usually requested for systems with stringent real-time constraints. This mechanism must however respect two conditions :

1/ the suspended actor must not be the actor which processes the interrupt-message : there cannot be preemption or parallelism between two processing-steps of the same actor, i.e. an actor cannot be suspended during a processing-step in order to process another message,

2/ priorities of processing-steps (through the associated ports) must be enforced.

3.3/ Inputs/Outputs

Just as in traditionnal systems, system actors offer to application actors a high level interface for device accesses.

For system actors in charge of handling I/Os, the physical devices are seen as ports; the request of an I/O is a message sent onto this port; the end of I/O is a message sent from this port.

For instance, a physical device D is represented by a port Pd (of the kernel). A system actor DMA (Device Manager Actor) in charge of managing D requests an elementary I/O on D by sending a message

Ps, Pd, [Request for an I/O]

This message contains the requested parameters for the I/O (read or write, address of a buffer, number of bytes, etc...). The kernel recognizes Pd as representing the physical device D and starts the elementary I/O operation; the end of I/O is signalled by an interrupt, transformed (by the kernel) into a message (see section 3.2)

Pd, Ps, [End of I/O]

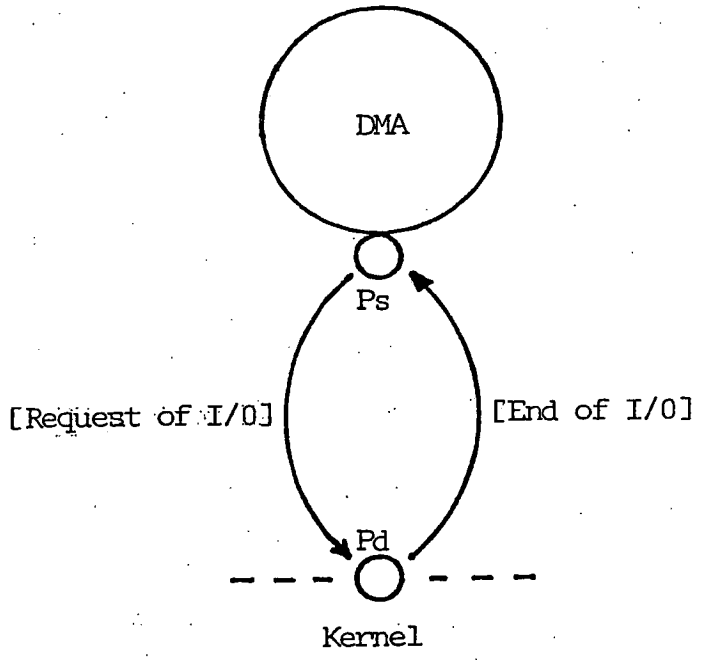


figure 3.4 : Implementation of I/Os

For system actors, the available I/O operations are the operations of the machine itself : for instance, I/O of one character on an asynchronous line, I/O of 128 bytes on a floppy disk, etc...

This organization of I/Os will allow easier implementation of CHORUS on multi-processor machines where I/Os are achieved by specialized processors : an I/O processor may be easily adapted in order to receive and send messages from or onto a port Pd; indeed, this concerns only the interface of the I/O processor and does not imply that this I/O processor runs CHORUS.

3.4/ Error_handling

Various errors may arise during operation of an actor :

- error in a service request (incorrect parameter, unauthorized request, ...),
- error in behaviour (sending an incorrect message, ...),
- error in an instruction (division by zero, addressing out of memory, ...).

CHORUS permits actors to attempt recovery from some of these errors according to the following scheme :

Errors are detected by system actors, by the kernel or by the machine; in the first two cases, the system actor or the kernel may send a diagnosis message onto a port specified by the actor which produced the error; in the last case, the kernel handles the error and transforms it into a message. In all cases, actors receive error reports as messages. The port which receives the diagnosis message does not need to be a port of the actor which produced the error; for instance, an actor A may debug an actor B and receive all diagnosis messages associated with B's errors.

However, if an actor does not want to attempt recovery from errors, it will be destroyed (in the case of an error, of course!...) with a diagnosis message sent to another actor.

3.5/ Naming

In CHORUS, each entity which needs to be designated (actor or port) is uniquely identified with a global_name which designates this entity from wherever in the distributed system; this designation is stable in space and time in the sense that a global name designates always the same entity wherever be the actor which uses that name and whenever it does it. This global name is created for the entity and will never be reused to designate another entity.

Global names are obtained as the concatenation of three fields :

site | type of name | local unique name

"Site" designates the site where the entity is created; as names of sites are all distincts, concatenation of the name of a site and of a local unique name gives a global unique name. Note that "site" designates the site of creation of the entity, not its site of residence : the entity may move from one site to another without changing its name.

"Type of name" makes it possible to distinguish several classes of entities. For instance, type of name may be "sedentary" or "nomadic" : a nomadic entity may migrate from one site to another whereas a sedentary one may not (this attribute is defined at the creation of the entity and may not change); this distinction facilitates the location of entities (for routing, for instance) : a sedentary entity may reside only on the site given by its name.

Basically, all designation mechanisms in CHORUS use global names : for instance, in communication, an actor designates both its ports and distant ports by their gobal names; a service is designated by the global name of the port which offers it; if an actor wants to destroy another actor, it designates it by its global name; etc... However, for performance reasons, accelarators may be built on top of this mechanism, which allow faster and cheaper designation (see section 4.2).

This large scope in the usage of global names is tempered by the protection mechanisms (see section 2.5) : an actor may request any service on any entity, but the service will be performed only if the protection mechanism authorizes it (remember that communication is also a service, requested on ports).

3.6/ Local and distant communication

The CHORUS system provides a communication service which is site independent (see section 2.3.1); section 3.1.1 mentionned that the kernel is in charge of local transport of messages. This section presents the complete scheme for local and distant transport of messages in the basic datagram mode.

Let an actor A send a message M "Pa, Pb, [Text of the message]". The kernel receives the message at the end of the processing-step of A. The kernel has a table of local opened ports (see section 3.1.3) :

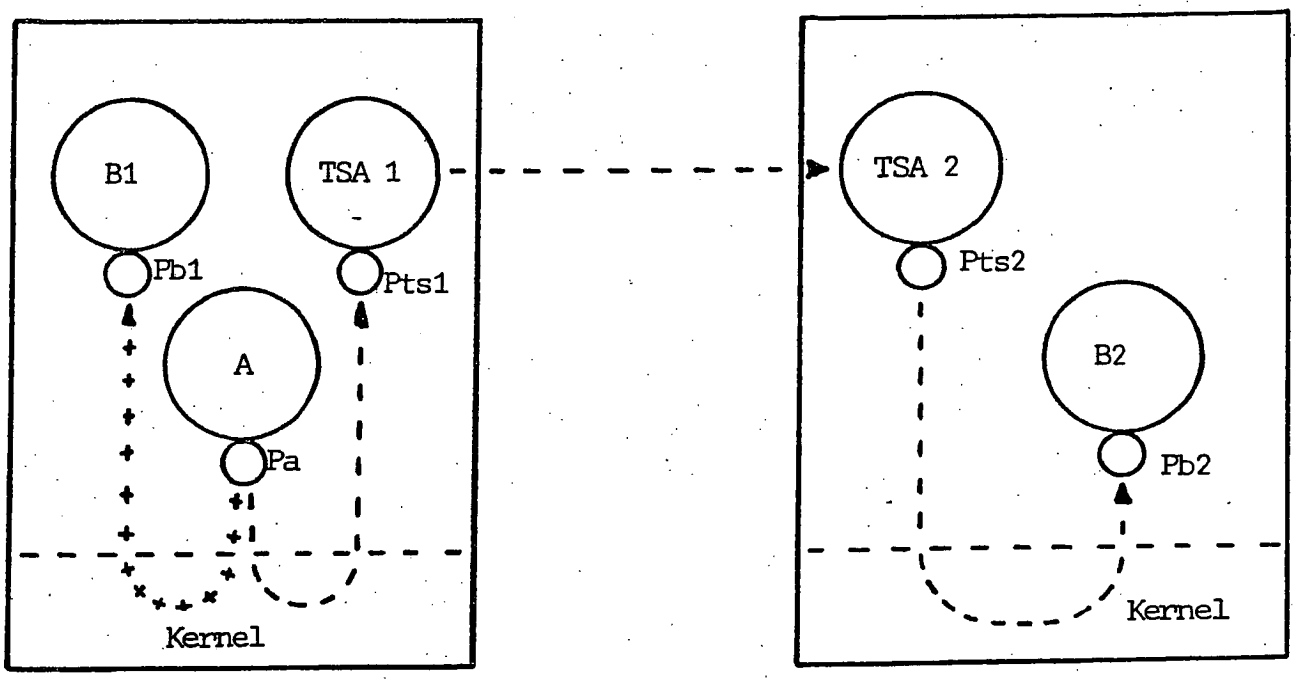
(1) If Pb is found in that table, the kernel performs the local transport of the message : the kernel places M in Pb's queue.

(2) Otherwise (i.e. Pb is opened and distant or closed or even nonexistant), the kernel places M in Pts's queue, where Pts is a "surrogate" port for all distant ports (namely, the port of the Transport Station Actor - TSA1) known from the kernel. M activates a processing-step in TSA1. TSA1 applies a routing algorithm (described below) in order to find the site where Pb resides; if this algorithm succeeds (i.e. if Pb is opened, distant and located on an accessible site), TSA1 transmits M through the communication medium to TSA2, the Transport Station Actor resident on the same site as Pb; if the algorithm fails, message M is lost. TSA2 sends "Pa, Pb, [Text of the message]" and the local kernel places M in Pb's queue.

Note that TSA1 receives M on Pts which is not the destination port of M and that TSA2 sends M through Pa which is not one of its own opened port. These two "derogations" are necessary to keep a uniform view of communications, whether local or remote : whatever the respective locations of Pa and Pb, actor A sends from Pa onto Pb and actor B receives also from Pa onto Pb. The kernel controls strongly usage of these derogations as it knows TSA to be the Transport Station Actor : only this actor is allowed to use these derogations.

Note also that if Pb is not an accessible opened port (i.e. either closed or nonexistant), M is lost and no diagnosis is provided; protocols built on top of the basic datagram facilty

would recover from this kind of error.



+++ local communication
--- distant communication

figure 3.5 : Local and Distant communication

Routing

In the current implementation, the Transport Station Actors (TSA) run a basic routing mechanism (described below) which has been designed to be very reliable even though not always optimized. The goal of this routing mechanism is to allow TSAs to follow the possible migrations of ports without managing large tables of ports.

Remember that this section describes the routing algorithm and transport protocol between the TSAs, not the transport protocol between application actors!

There is one TSA on each site. Each TSA manages three tables :

- the global names of all local opened ports (table A).
- global names of distant ports with their site of residence (table B). When a TSA starts its operation, this table is empty; update of this table is described below.
- the network addresses of other TSAs (table C) or a broadcast address if available.

The routing mechanism is as follows :

(1) When a TSA receives "Pa, Pb, [Text of the message]" from its local kernel, two cases may occur :

a/ Pb is in table B (see figure 3.6) : TSA sends the message over the network to the site of residence of Pb and waits for an acknowledgement.

- If TSA receives the acknowledgement, the communication is complete.
- If TSA does not receive any acknowledgement, it removes Pb from its table B and goes to case b/ (below).

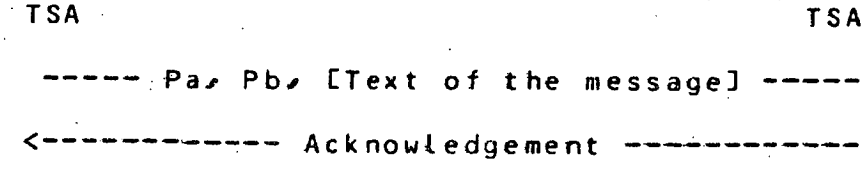


figure 3.6 : The distant transport protocol if Pb is in table B

b/ Pb is not in table B (see figure 3.7) : TSA interrogates other TSAs (using table C) in order to find out where Pb resides.

- If one distant TSA' answers positively, TSA sends the message

over the network to TSA' and updates its table B.

- If no distant TSA' answers positively, the message is lost.

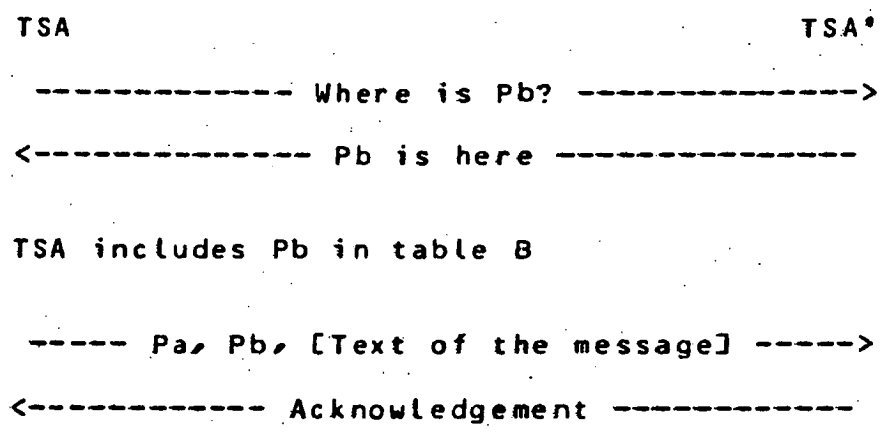


figure 3.7 : The complete distant transport protocol

(2) When a TSA' receives "Pa, Pb, [Text of the message]" from the network (i.e. from a distant TSA), two cases may occur :

a/ Pb is in table A, i.e. Pb is a local opened port (see figure 3.6) : TSA' sends the message locally and acknowledges it to the distant TSA.

b/ Pb is not in table A (see figure 3.8) : TSA' informs TSA that Pb is no longer on the local site (with a negative-acknowledgement); TSA removes Pb from its table B and goes to case 1-b/ (see above).

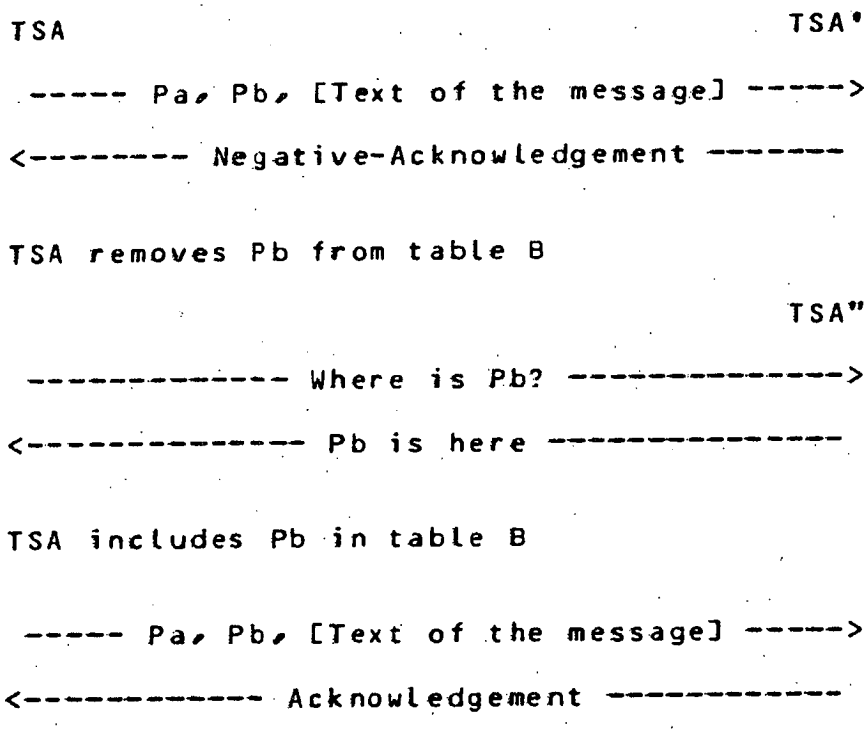


figure 3.8 : The distant transport protocol when Pb has migrated

Note 1 : messages sent over the network are numbered sequentially : so, the receiving TSA may detect duplication of messages.

Note 2 : a TSA uses time-outs in order to detect the non-reception of an acknowledgement which means the loss of a message or the failure of a distant TSA'.

Note 3 : table B (i.e. the table of distant ports) has a limited size. When it is full the least frequently used element is removed.

Note 4 : when a new site is created, each TSA updates its table C (i.e. the table of the network adresses of other TSAs), the new site becomes accessible as do all its ports.

4/ Discussion of CHORUS choices

This section discusses CHORUS choices; a comparison between CHORUS and other distributed systems may be found in [Guillemont 84].

4.1/ Synchronization

In distributed systems, synchronization cannot be ensured by a central unique authority as in centralized systems. Efficiency and robustness impose distributed synchronization mechanisms based on the exchange of messages : distributed processing is naturally driven by messages and exchange of messages is the basic synchronization tool.

CHORUS proposes an integrated view of synchonization and communication which is clearly separated from processing : the sending of a message is asynchronous (i.e. non blocking), the receipt of a message triggers a processing-step. This mechanism is compatible with distribution (as communication is site independent) and introduces the loosest coupling between actors : the processing-step which produces the message must be ended before the message is emitted and triggers another processing-step.

This mechanism favours "asynchronous programming", which is well suited for distribution but which is not offered by modern high level languages which rely on the "synchronous procedure call". However, this last synchronization scheme (procedure call) may easily be built on top of the CHORUS mechanism, as explained below.

External_procedure_call

An actor A requests the execution of a procedure (= a processing-step) in an actor B and waits until B completes it. More precisely, A and B use the "Request-Response" protocol (see figure 4.1) :

- actor A sends "Pa, Pb, [Request]" and stops,
- actor B processes [Request] and sends "Pb, Pa, [Response]",
- actor A resumes its operation as it receives [Response].

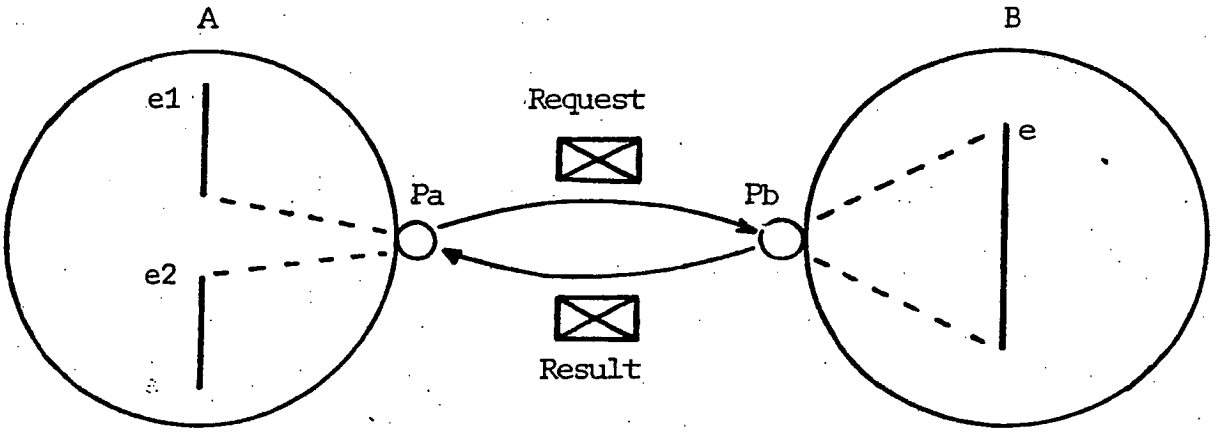


figure 4.1 : External procedure call

This synchronization is programmed as follows :

```
actor A                                  actor B

                                         .....
                                         RETURN (Pb, Select, [(Pa, Pb)];
                                                 Pb, Switch, [e];
                                                 .....);

ENTRY-POINT e1;
.....
RETURN (Pa, Pb, [Request];
        Pa, Select, [(Pb, Pa)];
        Pa, Switch, [e2]);

            --- Pa, Pb, [Request] --->

                                         ENTRY-POINT e;
                                         .....
                                         {processing of [Request]}
                                         {constitution of [Response]}
                                         .....
                                         RETURN (Pb, Pa, [Response];
                                                 .....);

            <--- Pb, Pa, [Response] ---

ENTRY-POINT e2;
.....
```

figure 4.2 : Programming the external procedure call

At the end of the processing-step e1, A sends the [Request] onto Pb; the selection condition given by A means that only messages sent from Pb onto Pa (i.e. [Response]) may be selected; the switch condition means that the selected message received onto Pa must trigger the processing-step e2.

Therefore, this realizes the "external procedure call" : A executes e1, requests some processing-step in B (here "e") and resumes in e2 as the response is received.

Note : this way of programming may be risky as A is indefinitely blocked if B fails before sending the response message. In order to avoid that situation, the correct programming of A is

```
RETURN (Pa, Pb, [Request];
        Pa, Select, [(Pb, Pa)];
        Pa, Switch, [e2];
        Pa, Time_Out, [Pb, Delay]);
ENTRY-POINT e2;
```

If B fails, the time-out on Pa will trigger A in e2.

This particular sequence of code which allows two processing-steps to be linked sequentially in an actor has been named EP_CALL (for External Procedure CALL). Actor A may thus be programmed as follows :

```
ENTRY-POINT e1;
.....
.....  (processing-step e1)
.....
EP_CALL (Pa, Pb, [Request], Delay);
.....
.....  (processing-step e2)
.....
```

Of course, this is particularly well suited for programming actors in Pascal for instance : the expression of a call to an external procedure is similar to the expression of a call to an internal procedure.

4.2/ Designation

The designation mechanism used in CHORUS relies on the usage of global names (see section 3.5) This has several advantages :

- it is reliable in the sense that there cannot be any misunderstanding in the relation global name/entity.
- there is no need of any name conversion or mapping neither between actors nor between sites.

Here again, the choice in CHORUS is the most basic and secure mechanism : other mechanisms, faster or more specific, may be built on top of this one (but not vice versa); and if these mechanisms fail, the system must be able to go back to the basic designation of entities in order to recover.

Examples of two other designation facilities are given below.

Accelerators

This mechanism is intended to be cheaper and faster than the usage of global names.

An actor may define local names which designates

- its own ports.
- liaisons between its ports and distant ports (= couples of ports).

Local names are understood only by the actor and the local kernel. They accelerate the research of ports in routing tables, controlling and queuing messages. However, messages transported from one port onto another will be associated with the global names of their ports.

Functionnal_names_and_group_names

A system actor, the Name Manager Actor, offers the possibilty to define two kinds of names [Senay 83] :

- FUNCTIONNAL NAMES (1 among n). Several ports in an application may offer equivalent services; when an actor requests this service, any of these ports may process the request message. These ports have, in addition to their global name, a common functionnal name Pf : according to the scheme given in section 3.5, the "type" of Pf is "functionnal name". When an actor requests the service, it sends its message onto Pf; the local TSA (Transport Station Actor) receives the message, locates one port (possibly local) with that name and sends the message onto that port. The location of that port is acheived by using the protocol presented in section 3.6 : the TSA broadcasts the message [Where is Pf?] and retains only the first answer (which is not registred

in its table of distant ports).

This mechanism is also used when there is exactly one port per site for the same service and when an actor must always communicate with the local port (it is the case for most system services) : all these ports have the same functionnal name and a message sent to Pf reaches always the port Pf which resides on the same site as the sender of the message. This simplifies dynamic "linking" between an actor and the system services it uses.

- GROUP NAMES (n among n). An actor may request the broadcasting of a message to a set of ports. This set of ports has a group name Pg : indeed, no port in the system is named Pg; Pg only represents a group of ports and Pg has the structure of a port name. According to the scheme given in section 3.5, the "type" of Pg is "group name". Each NMA (Name Manager Actor) manages a partial list of ports belonging to the group. When an actor sends a message onto Pg, the local TSA receives the message; it requests from the local NMA the local list of ports belonging to Pg and it sends a copy of the message to each port in the list; simultaneously, it broadcasts the message to all other TSAs and each TSA sends also a copy of the message to all ports in the local list of the Pg group. Note that this mechanism is recurrent : a list of ports belonging to a group may contain the name of a port with a group name.

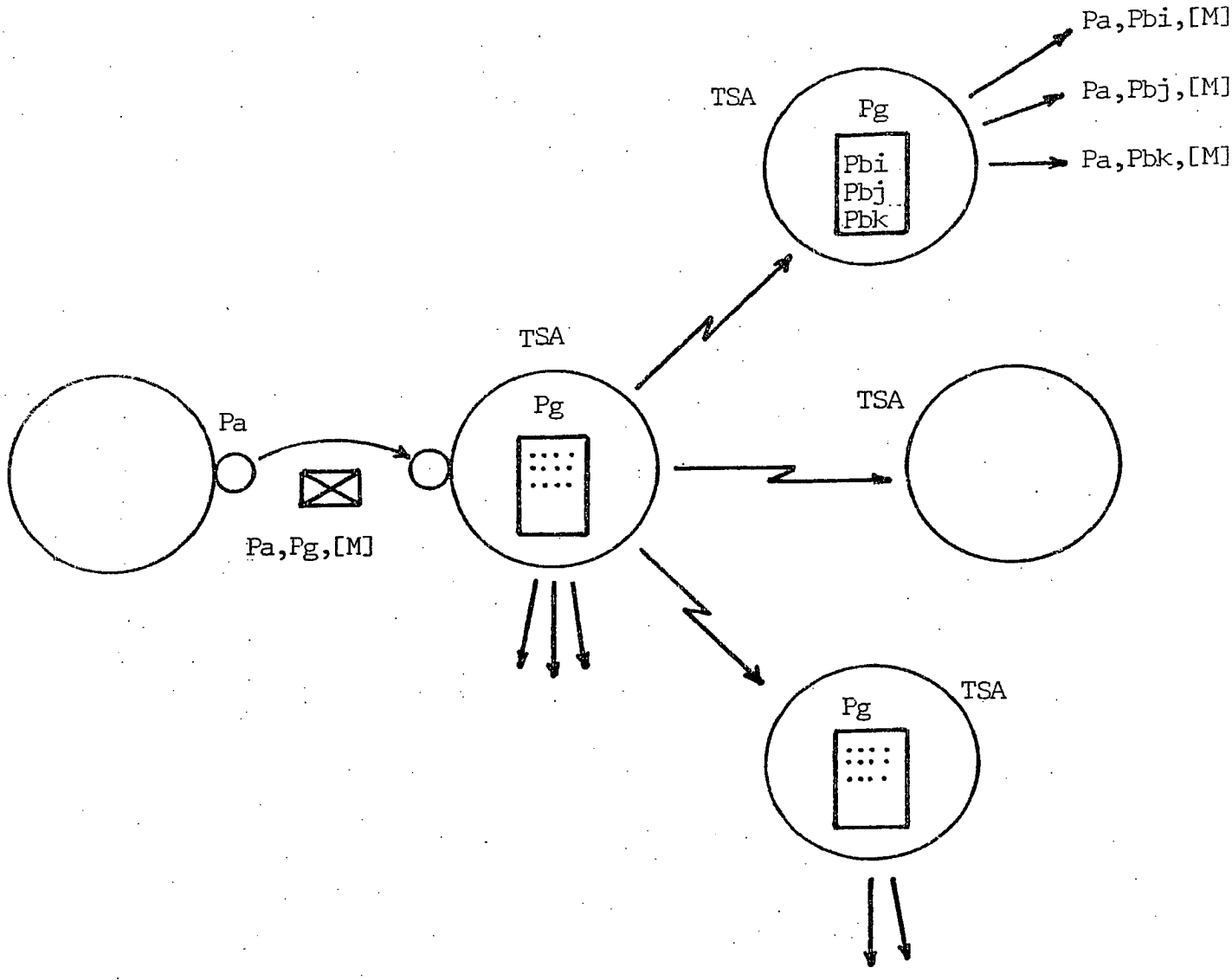


figure 4.3 : Group names

4.3/ Protection

Distribution places stringent limits to the amount of control possible at compile-time and clearly requests additional controls at run-time. The absence of central control tends to make distributed systems more complex and impose robust protection mechanisms or additionnal encryption in order to restrict propagation of errors and failures. However, the traditional capability scheme requires additional hardware which is usually not available, even on modern computers. This suggests that a simpler scheme might be more adequate for simple distributed systems which have the additional capability of physical isolation.

The protection scheme adopted in CHORUS consists in two levels (see section 2.5) :

- the system controls that the architecture is enforced (i.e. mainly controls the relation actor/port).
- control procedures describe other controls for each application.

The first level of protection allows, for instance, an actor to base its defence on a list of authenticated ports which are authorized to communicate with it; the actor behind a port cannot change unless the port is closed and reopened; but this last system service is controlled by the second level of protection.

In communication, the send and receive control procedures associated with the source and destination ports make it possible to check passwords in messages, to encrypt and decrypt messages, to check types (if they are apparent in messages), etc...

Control procedures are written by users : the nature of the control may be adapted for each application. This allows for

instance control procedures and application code written by different teams. It also allows for instance many controls in the debugging phase of an application and less control, and thus less overhead, in the operational phase. CHORUS imposes only the list of possible control procedures and the moments when they are executed.

4.4/ Reconfiguration

Operation of an actor is a sequence of processing-steps. During the execution of a processing-step, an actor neither sends nor receives any message : one message triggers the processing-step and messages are sent only at the end of the processing-step. Therefore, when an actor receives a message, it can be sure that the proccessing-step which produced that message has correctly ended; conversely, if a processing-step fails, no message is sent.

Based on that remark, a simple reliability mechanism has been developped ([Banino 82], [Fabre 82]) : the basic idea of this mechanism is that a reliable service is performed by a couple of identical distributed actors (the Master and the Slave); the Master receives the Request messages and the Slave receives a copy of these messages forwarded by the Master; so, both actors receive the same request messages, in the same order, and both execute the same processing-steps, in parallel; but only one actor (the Master) sends the response message. So, both actors stay always identical and the client of the service receives one response. If the Master fails, the Slave becomes the Master exactly at the beginning of the processing-step in which the Master failed :

indeed, if the Master has failed, it did not send any of the messages prepared during its last processing-step; as the Slave becomes the Master at the beginning of that processing-step, there is no loss nor duplication of messages from the client's point of view : the new Master has received a copy of the request message - while it was Slave - and it will send the response as it is now Master.

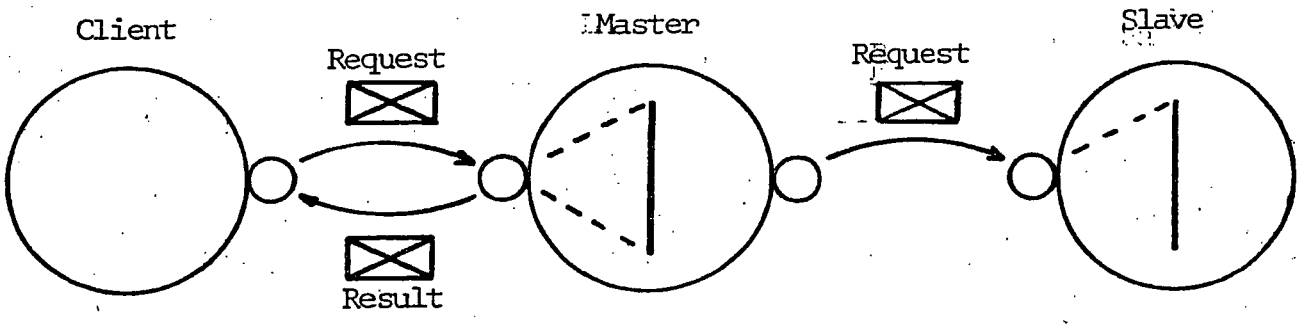


figure 4.4 : Distributed coupled actors

4.5/ Heterogeneity

The CHORUS architecture is independent of the hardware on which it is implemented and also independent of the communication medium which connects the sites. The CHORUS architecture relies on a kernel resident on each site which offers a standard interface. Therefore CHORUS may be implemented on a variety of machines : each machine has a specific kernel and some system actors have to be adapted too (e.g. the Memory Manager).

However some remarks are due concerning heterogeneity :

(1) Communication between different machines must take care of the internal coding of messages : order of high and low bits

in a byte, order of bytes in a word, number of bytes in an integer, etc...

(2) A set of CHORUS machines may as well be connected to a non CHORUS machine, provided this last machine is seen as a set of ports sending and receiving messages. In that case, send and receive control procedures will be very useful for protection. For instance, in a multi-processor machine, some processors may support the CHORUS architecture whereas others (the I/O processors, for instance) may not : the processors of the second goup may however send and receive messages on ports.

(3) If CHORUS is machine dependent, performance is not : it is clear, for instance, that broadcasting through an Ethernet-like network should be more performant than through a meshed long-haul network! In other words, the broadcasting facility may be available whatever be the communication medium but the users must be aware that performance depends on the implementation.

4.6/ Software engineering

Figure 2.6 presented a representation of a processing-step close to the operation of the actor. This representation is too rigid in order to be taken as a software tool for programming actors.

Actual implementations of CHORUS are written in Pascal. In order to write models of actors in Pascal, CHORUS developped a set of tools which help programming. These tools are summarized below :

1/ a model of actor is described as a Pascal program where each processing is coded as a procedure. Standard compilers are used in order to compile models of actors.

2/ a model of actor defines three standard variables, say "Source_Port", "Destination_Port" and "Message" which contain, at the beginning of each processing-step, the description of the message received; these three variables are overwritten at the beginning of each new processing-step.

3/ a model of actor defines (possibly implicitely) a stack "Queue" where all messages which have to be emitted at the end of the processing-step are queued, and standard interface procedures queue the messages prepared by the actor and sometimes build them too. Here are examples of interface procedures :

Prepare (P, Q, M)

queues "P, Q, [M]"

Select (Q, P)

builds and queues "P, Select, [(Q, P)]"

Switch (P, E)

builds and queues "P, Switch, [E]"

Time_Out (Q, P, Delay)

builds and queues "P, Time_Out, [Q, Delay]"

Create_Port (P, Model, Name, Initial parameters)

builds and queues "P, Create_Port, [Model, Name, Initial parameters]"

etc...

4/ RETURN is an interface procedure without parameters. When the actor calls RETURN, control is given back to the kernel and the "Queue" of messages is transmitted to it.

According to these rules, a processing-step and a model of actor look like :

```
PROCEDURE entry;
.....
IF ... THEN Time_Out (Q1, P1, Delay)
       ELSE Prepare (P2, Q2, Mess);
FOR i := 1 TO n DO Prepare (Pi, Qi, Mi);
.....
RETURN;
END;
```

figure 4.5 : Processing-step

```
PROGRAM model_of_actor;

PROCEDURE entry_1;
.....
END;

PROCEDURE entry_2;
.....
END;

etc...
```

figure 4.6 : Model of actor

The first set of interface procedures presented above builds and queues single messages.

A second set of interface procedures are also available which perform more complex functions. In particular, the "external procedure call" (see section 4.1) is offered as a procedure EP_CALL; based on that procedure, CHORUS offers a set of interface procedures which request synchronously all system services. For instance, the procedure Synchronous_Open_Port requests synchronously the opening of a port; this procedure is programmed as follows :

```
Synchronous_Open_Port (Port, Name, Priority, VAR Diagnosis);

BEGIN
Save "Source_Port", "Destination_Port", "Message";
Save "Queue";
EP_CALL (Port, Open_Port, [Name, Priority], Delay);
(* After the execution of that procedure, informations about
   the message received are in "Source_Port", etc...  *)
IF Source_Port = Time_Out
     THEN Diagnosis := "Delay Elapsed"
     ELSE extract Diagnosis from Message;
Restore "Queue";
Restore "Source_Port", "Destination_Port", "Message";
END;
```

The first set of interface procedures may be called "asynchronous interface procedures", the second set "synchronous interface procedures". This second set is very convenient for

programming in languages (like Pascal) based on the procedure call : a system service is requested by calling a synchronous interface procedure.

A complete set of interface procedures is given in appendix.

5/ Illustration of the CHORUS architecture

This section is built around an example of a simple distributed mail system. This example should in no case be taken as representing our view of the distributed mailing; it is only taken here to show some problems and to illustrate the CHORUS approach. For this purpose, the examples have been kept voluntarily simple or even naive.

5.1/ Sketch of the mini-mail system

The mini-mail system consists of actors of two models :

- the mail servers MS are permanent actors, one per site.
- the mail users MU are temporary actors, one per user; they may appear on any site.

Each (human) user is associated with a mailbox.

Each mail server actor

- manages a set of mailboxes and messages for each mailbox,
- watches over another mail server MS'; if MS' fails, messages in mailboxes managed by MS' are temporarily unavailable (they may have been saved on stable storage), but MS creates "copies" of mailboxes managed by MS' so that they may receive new messages; these mailboxes remain available though messages received before the failure are no more accessible.

A mail user actor MU is created for each user logged in the system who wants to use the mini-mail system. A mail user actor may

- send a message in another user's mailbox,
- read messages in its own mailbox.

Each mailbox is represented by a port; the mailbox of user U is designated as the port MbU (Mailbox of user U). The mini-mail system is seen by mail users as a set of ports MbU; each user knows the name of its mailbox and communicates it to its friends. The ports MbU are ports of the mail servers; the mail server which has opened the port MbU manages the mailbox for user U.

The mail servers have also private ports PpS (private Port of server S) whose names are not known from users. Even if a mail user would know a PpS, the receive control procedure at PpS would reject the messages it could send (see section 5.4).

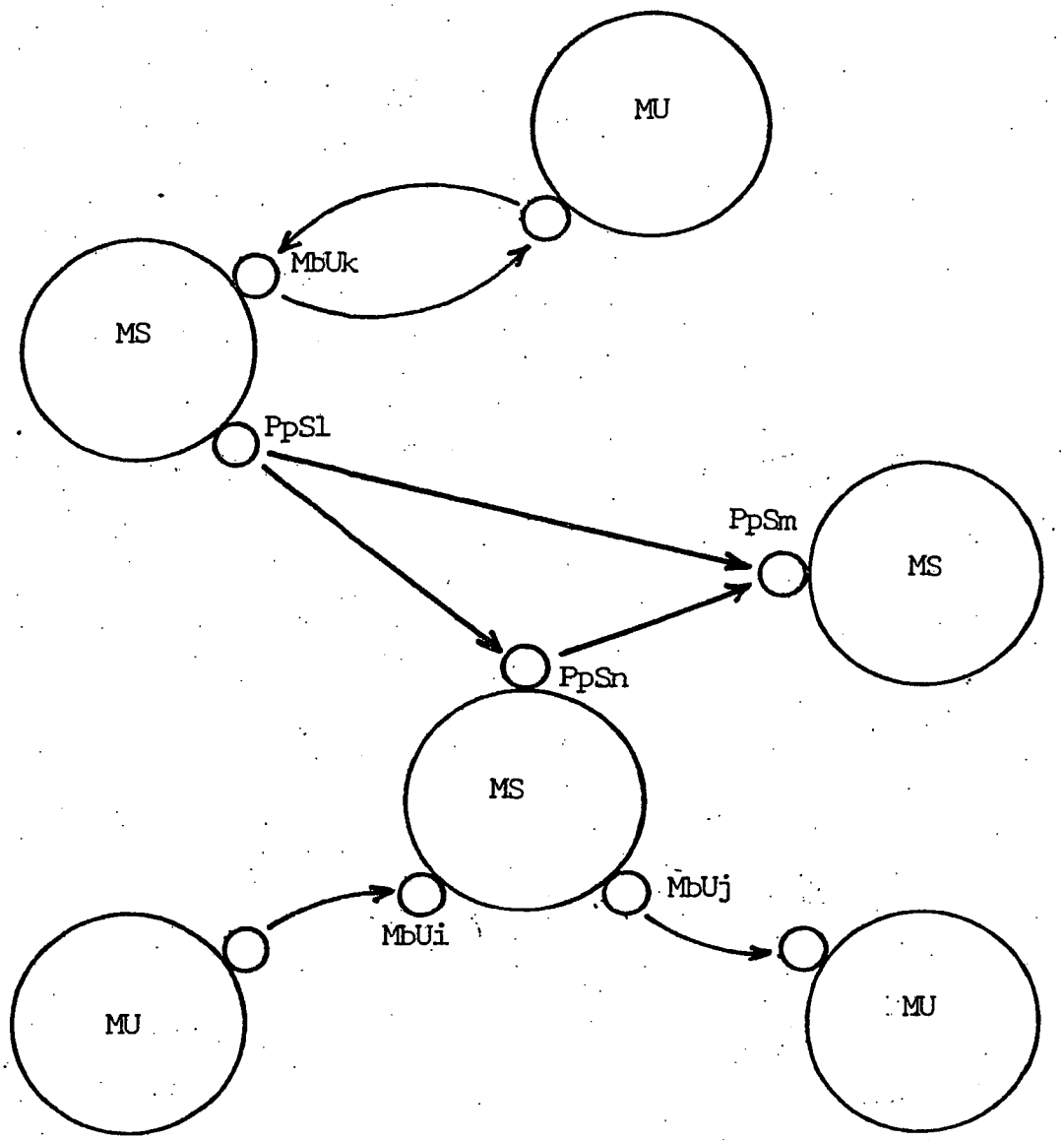


figure 5.1 : The mini-mail system

Using_mailboxes

(*) A mail user actor MU requests the creation of a mailbox for a user by sending the request

Port, Create_Mailbox, [Create, User, Password PwU]

Create_Mailbox is a functionnal name for ports of the MSs. The mini-mail system creates a new unique name MbU for the mailbox and a mail server replies

Create_Mailbox, Port, [MbU]

(*) A mail user actor sends a message into a mailbox MbU by sending

Port, MbU, [Send, Message]

(*) A mail user actor reads a mailbox MbU by sending

Port, MbU , [Read, Password PwU]

the mail server behind MbU replies

MbU, Port, [Read, Message]

(*) A mail user actor requests the destruction of a mailbox by sending

Port, MbU, [Destroy, Password PwU]

5.2/ Synchronization

Each mailbox is managed by a mail server : the absence of parallelism within an actor provides mutual exclusion for accesses to each mailbox; two mail users may send simultaneously two messages in MbU, their messages will be recorded sequentially in the mailbox; similarly, a mail user may read its mailbox and another mail user may send a message to it, their requests will be processed sequentially.

5.3/ Designation

Each mailbox is designated with a global unique name MbU. This name may be derived from the user's name (like in Multics, for instance). These names are site independent; the ports are nomadic.

Access to a mailbox does not make any assumption on the localisation of the mail server which manages it. In the same way, a mailbox may migrate from one mail server to another (see section 5.5) without any change in the mailbox's access.

5.4/ Protection

Protection applies on mailboxes and on mail servers.

Protection of mailboxes

Each mailbox is created with a password PwU (see section 5.1). The port MbU is given that PwU when it is created (see section 2.3 - creation of a port). All messages for accessing that mailbox are received onto MbU. The receive control procedure associated with MbU protects the mailbox :

- messages which do not correspond to a known operation (send, read, destroy, etc...) are rejected.
- messages (except for send) which do not contain PwU are also rejected.

Protection of mail servers

An analogous control mechanism protects the mail servers.

The mail servers communicate through private ports PpS; each PpS is associated with a password PwS.

When a mail server is created (by another mail server) it receives in its initial message the list of the other couples (PpS'/PwS'); the new mail server creates and opens its own PpS and sends onto each PpS'

PpS, PpS', [my PwS, your PwS']

this message authenticates PpS for other mail servers and transmits the associated PwS. So, recurrently, the mail servers constitute a group of authenticated ports PpS.

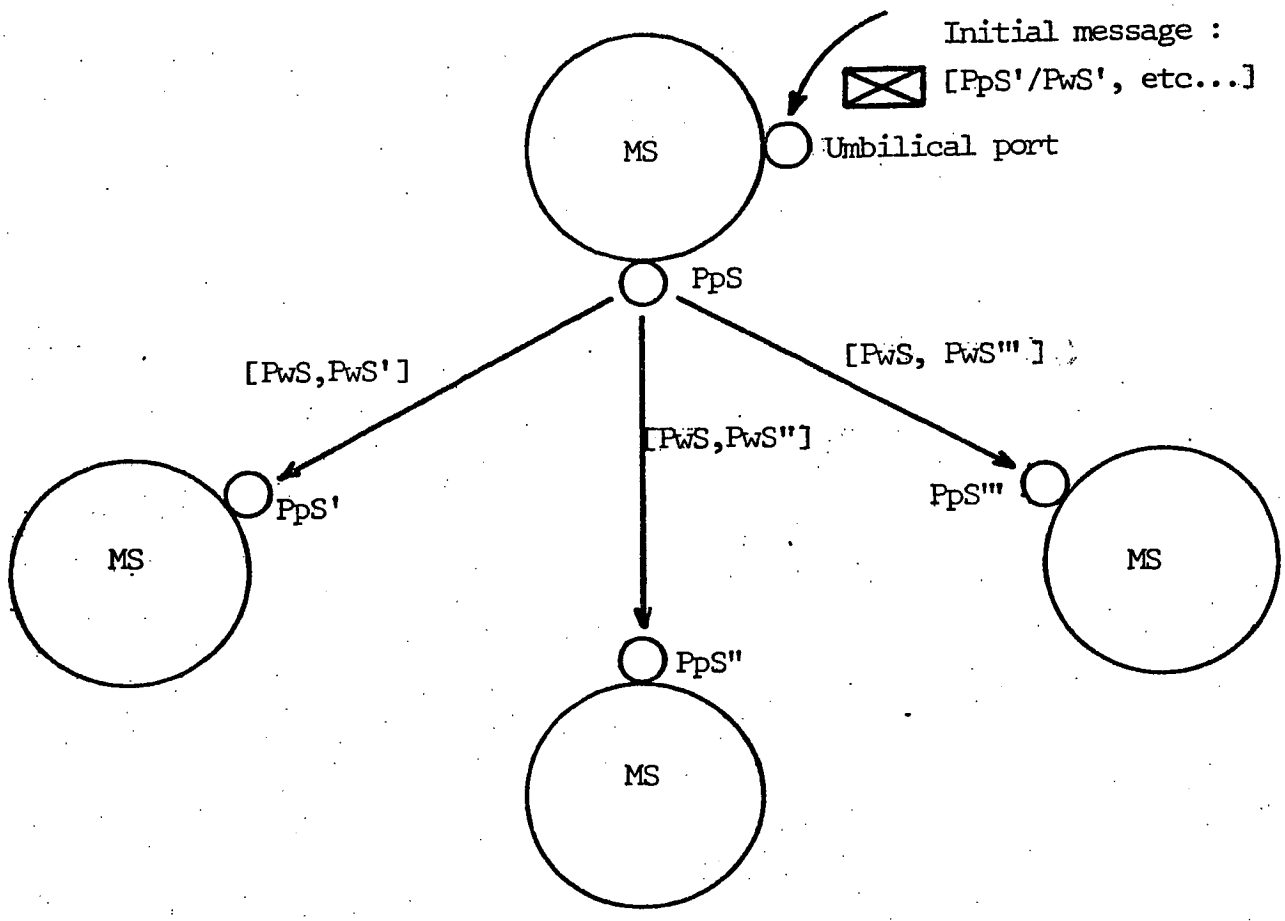


figure 5.2 : Authentication of PpS

Each further message sent onto PpS must

- be sent by one of the PpS',
- contain PwS.

This last control may be performed by the receive control procedure of PpS.

The set of mail servers constitutes an activity. Protection and reconfiguration (see section 5.5) are designed on the base of this activity.

5.5/ Reconfiguration

Reconfiguration is based on

- mutual watching of mail servers,
- migration of ports MbU.

Let MS1 be a mail server watching over MS2; MS2 manages mailboxes MbU1, MbU2, ..., MbUp. Reconfiguration may be sketched as follows :

1/ MS2 sends periodically to MS1

PpS2, PpS1, [list of MbUi, PwS1]

2/ MS1 reenables a time-out on the receipt of each message from PpS2 onto PpS1

P, Time_Out, [PpS2, PpS1, Delay]

3/ If MS2 fails, the time-out elapses and MS1 detects the failure of MS2. As MS2 has failed, all its ports, and in particular all MbUi, have been automatically closed by the system. MS1 opens these ports

PpS1, Open_Port, [MbUi, Priority]

For external users, mailboxes MbUi remain available and their acess remains unchanged. The only difference is that messages received by MS2 for MbUi are (temporarily) unavailable.

4/ When MS2 reappears, MS2 sends a message to MS1; MS1 closes the ports MbUi and sends to MS2

PpS1, PpS2, [list of MbUi, list of messages received on each MbUi, PwS2]

MS2 re-opens the MbUi and the service is again completely available.

Using analogous mechanisms, the mail service may be extended by adding new mailboxes or even by creating new mail servers.

5.6/ Protocol_programming

Example of flow control without anticipation between a mail server and a mail user :

A mail user actor wants to read on its port "Port" all messages in its mailbox MbU. The mail server and the mail user work as coroutines :

- the mail server sends a message and waits for its acknowledgement to send the next message.
- the mail user sends an acknowledgement and waits for the next message.

Mail server :

```
ENTRY-POINT e;
IF end_of_mailbox THEN M := End_of_mailbox
ELSE M := Next_message;
RETURN (MbU, Port, [M];
        MbU, Select, [(Port, MbU)]);
```

```
Mail user :

ENTRY-POINT e;
IF M = End_of_mailbox THEN ...;
processing of M;
RETURN (Port, MbU, [Acknowledgement, PwU];
        Port, Select, [(MbU, Port)]);
```

In order to recover from possible loss of messages, mail server and mail user may be programmed as follows : the algorithm is the same as above, but actors enable time-outs on their ports and check wether the time-out has elapsed or not when they execute (if Source_Port = Time_Out, the message processed is an indication of time-out elapsed; if Source_Port <> Time_Out, the message processed is a normal message).

```
Mail server :

ENTRY-POINT e;
IF Source_Port = Time_Out THEN M := last_message_sent
ELSE IF end_of_mailbox THEN M := End_of_mailbox
     ELSE M := Next_message;
     last_message_sent := M;
RETURN (MbU, Port, [M];
        MbU, Select, [(Port, MbU)];
        MbU, Time_Out, [Port, Delay]);
```

```
Mail user :

ENTRY-POINT e;
IF Source_Port <> Time_Out THEN
     IF M = End_of_mailbox THEN ...;
     ELSE processing of M;
RETURN (Port, MbU, [Acknowledgement, PwU];
        Port, Select, [(MbU, Port)];
        Port, Time_Out, [MbU, Delay]);
```

Note that, for the mail user, even if the processing-step is triggered by a time-out, the actor sends an acknowledgement : this allows the recovery of the previous acknowledgement from a possible loss; this implies also that a mail user must be able to detect the reception of two consecutive identical messages (messages are numbered sequentially).

6/ Implementation_of_CHORUS

CHORUS has been implemented on a set of Intel 8086 connected through Danube [Naffah 80] an Ethernet-like broadcast network.

A second implementation is in progress on a set of SM90 [Finger 82], a multiprocessor machine based on the M68000.

Both implementations are written in Pascal.

6.1/ Implementation_on_Intel_8086

This implementation uses the UCSD-Pascal development system [Softech 80]. The compiler produces P-code which is interpreted. The compiler has not been modified; only the interpreter has been slightly changed in order to allow

- the sharing of memory and CPU between several actors and the kernel (which are all independent Pascal programs) in the same machine - the original interpreter works on only one program -,
- communication between the actors and the kernel.

This implementation has been realized as a test-bench for CHORUS architecture. It has proven the validity of CHORUS concepts and has shown that programming of actors was easy.

6.2/ Implementation on the SM90

A SM90 is a multi-processor machine and consists of

- a set of micro-processors Motorola 68000 called processing-units (PU) connected both to a global bus (common to all PUs) and to a local bus (private for each PU) through a Memory Management Unit (MMU),
- a set of memories connected either only to the global bus (Common Memory - CM) or both to the global bus and one local bus (External Local Memory - ELM) or only to a local bus (Local Memory - LM),
- a set of micro-processors Intel 8080 or Intel 8086 called exchange units (EU) connected to an ELM through a local bus. These EU are in charge of handling I/Os an various devices (terminals, disks, local network, etc...); an EU may access only its own ELM.

A CM is accessible to all PUs throuh the global bus. An ELM is accessible through a local bus by one PU and through the global bus by all other PUs. A LM is accessible through the local bus by one PU.

The maximum configuration of an SM90 is 8 PUs, 16 EUs and 16 Mega-bytes of memory (for all CMs, ELMs and LMs).

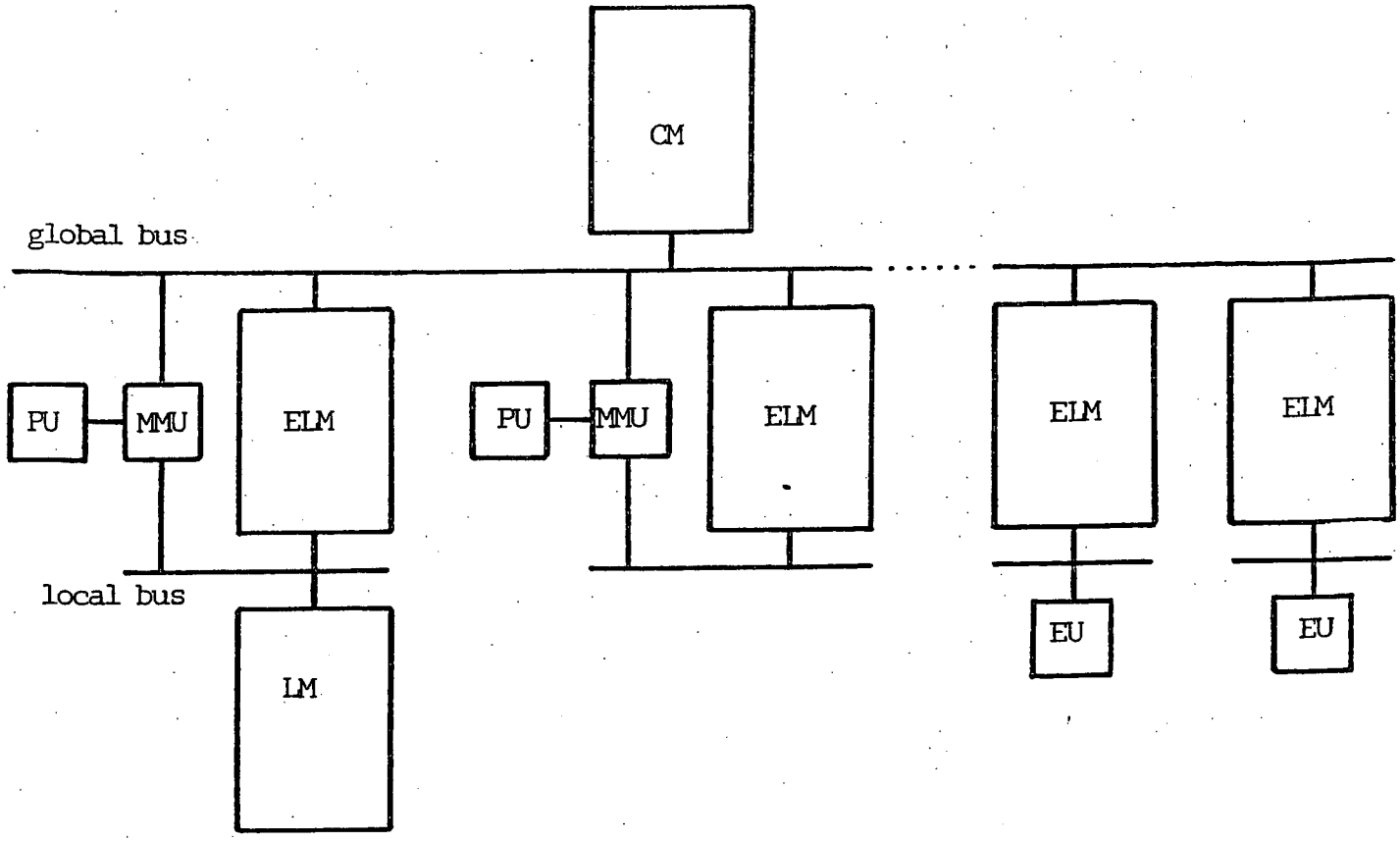


figure 6.1 : PUs, Memories and buses of the SM90

In the implementation of CHORUS on SM90, we define a CHORUS site as the set of ELMs, LMs, and PU connected to one local bus.

ELMs and CMs are used as a fast communication medium for exchanging messages between two sites in the same SM90. Communication between two sites is always through the exchange of messages whatever their respective locations (on the same SM90 or on two SM90s).

The Memory Management Unit (MMU) is an essential feature of the SM90 : it maps the addresses produced by one PU (considered as logical addresses) into physical addresses.

There is one MMU for each PU. Each MMU manages a set of 1024 segment registers and one Base-Length register (BL) : each

segment register describes a physical segment in CM, ELM or LM (physical address of the segment, length of the segment, protection). Each process uses a "window" of contiguous registers defined by the first register (Base) and the size of the window (Length); base and size of the current process are kept in the BL register; a logical address is interpreted by MMU as a segment number relative to B and a displacement in this segment.

The address decoding delay of the MMU is 50ns.

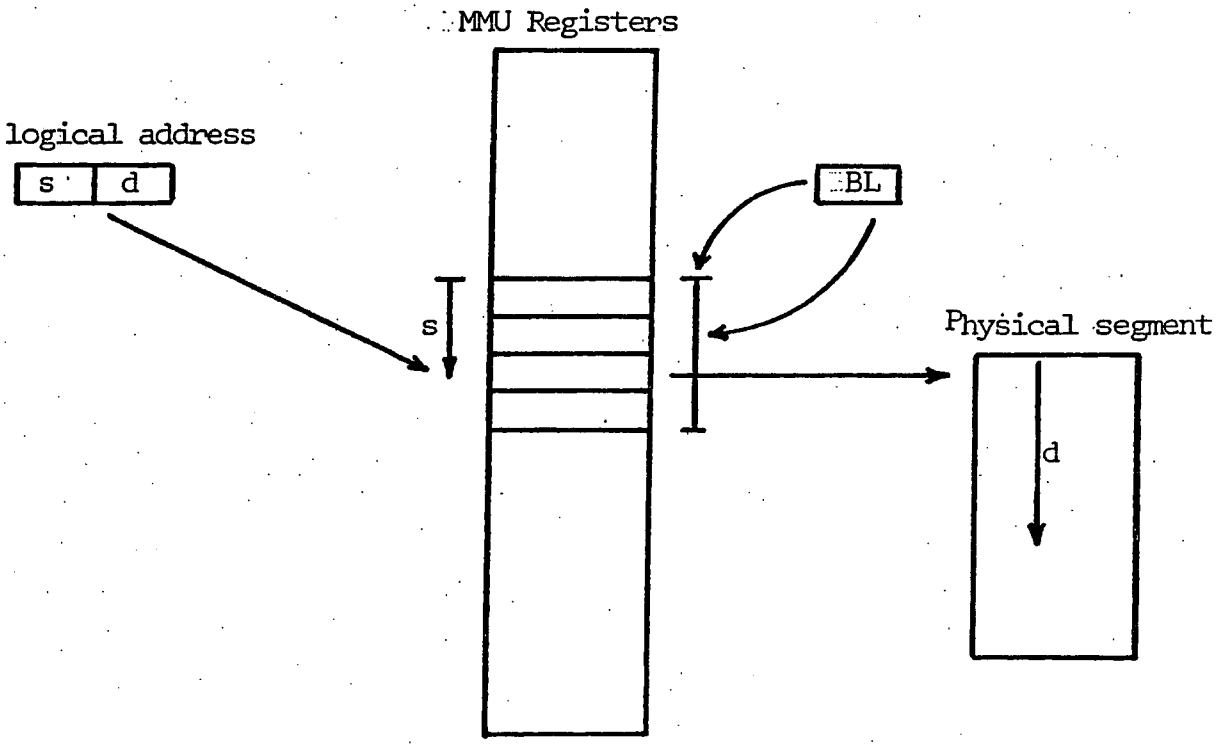


figure 6.2 : Translation of addresses

This MMU allows an increase in performance :

- the context switching is almost reduced to changing the BL register from the window of the current actor to the window of the next actor;
- a message is a segment; passing a message (on one site) consists

in tranferring a MMU segment register from one window to another. If the message is in ELM or CM, its MMU register may even be transferred from one site onto another in the same machine without copying the message itself;

- sharing code or data between actors is achieved by installing two identical registers into two differents windows;
- the great number of MMU registers (1024 and soon 2048) prevents registers swapping;

The identification of a message and a physical segment allows to transfer code, data, files... between actors exactly as any ordinary message, which will be very usefull for system actors.

The implementation of CHORUS on SM90 will focus on the measurement of the efficiency of CHORUS and will experiment the features of the SM90 for distributed systems.

6.3/ Using_Pascal

Our experience in the usage of Pascal [Guillemont 82b] led us to two conclusions :

1/ It is possible to write actors and most of the kernel with Pascal.

2/ Pascal is certainly not the appropriate language for CHORUS (neither for the system nor for applications).

Both implementations use the native compiler without any modification, but both use also some specificities of each Pascal. For instance

- in the UCSD Pascal, it is possible to define segments in a

program and the links between segments are dynamically installed at run-time; this has been used for communication between actors and the kernel (the kernel is a segment common to all actors).

- in the Pascal of the SM90, separated compilation is available; this is used in order to give actors some specific run-time structure in order to fulfill with the MMU of the SM90.

On the other hand, many features lack in Pascal for it to be the CHORUS language. For instance

- Pascal imposes that the sequence of execution of a program is driven by the program itself (with the IF, WHILE, FOR, ...). In CHORUS, the sequencing of processing-steps is determined by the messages received along with select and switch conditions : the set of processing-steps in an actor cannot be programmed as a pure Pascal sequential program.
- CHORUS proposes a construction of an actor in three steps (see section 2.6); the second step consists in the mixing of several modules in one model of actor; this is a link edition of different programs, which is prohibited in the standard Pascal.
- Protection should rely on common declarations of types (of messages, for instance) between various actors. Pascal does not provide that.
- Communication in Pascal relies on the procedure call. Comuniation in CHORUS relies an asynchronicity between the sender and the receiver actors.

7/ Conclusion

The CHORUS architecture is built with a small set of powerful concepts. In the various aspects of the architecture, we always did choose simple and basic mechanisms, on which a large variety of more peculiar strategies may easily be built. So, we did obtain a small kernel and a small set of system actors.

Both implementations and experimentation led us to the conviction that CHORUS is a sound basis for

- learning and familiarity with distribution,
- new researches and experiments.

The CHORUS architecture has been presented to a large public in industry, university, research centers : everybody agrees on the clarity and simplicity of concepts which make CHORUS attractive. People who intend to design distributed applications in terms of actors learn CHORUS mechanisms within half of a day!

With regard to new researches and experiments, some work is already undertaken in various fields, namely

- designation : an implementation of various strategies for managing functionnal names and group names is achieved and currently used.
- reliability : the first experiment of distributed coupled actors [Fabre 82] is going on and other strategies will be explored.
- real applications are being built upon CHORUS; they will give us more experience in the usage of CHORUS.
- language and software engineering for distributed applications

are going to be explored, with a special attention to the expression of distributed activity.

8/ Acknowledgements

It is a pleasure to thank all friends who contributed to this paper :

Authors, Friends, [Thank you]

APPENDIX

CHORUS Programming Interface

This appendix presents a set of procedures which constitutes the programming interface of CHORUS in the actual implementations in Pascal. As outlined in section 4.6, these procedures are divided into two subsets :

- the "asynchronous" interface procedures build and queue messages on a stack which is transmitted to the kernel at the end of the processing-step.
- the "synchronous" interface procedures request synchronously system services using the EP_CALL mechanism (see section 4.1).

This appendix gives the interface procedures related to the control of operation of an actor and to the manipulation of actors, ports and interrupts; the interface procedures related to the manipulation of physical devices, files, etc... are much dependent on the machine on which CHORUS is implemented : they are not given here.

In CHORUS, an actor may request simultaneously the same system service for several entities : e.g., an actor may send simultaneously several messages for the creation of several ports. These messages are processed in an unpredictable order and therefore responses come back also in an unpredictable order.

On the other hand, a system actor may receive request messages for various services on the same port : e.g., the File Manager Actor receives on the same port requests for reading and writing files.

These two considerations led us to adopt the following structure for the text of all request and response messages of system services :

Service Code

User Code

Parameters

- the "service code" precises which service is requested or performed.
- the "user code" is filled by the requestor actor; the system actor does not use it but copies it in the response message. Therefore, the requestor actor may associate the corresponding response and request messages : they both contain the same "user code".
- parameters depend on the service.

The "asynchronous" interface procedures include a "Local port" parameter as the "synchronous" interface procedures do not : this "Local port" is the port of the actor through which the request message is sent and on which the response message is received (see section 2.1.4); "synchronous" interface procedures always use the umbilical port of the actor in order to send and receive the messages.

The "asynchronous" interface procedures include also a "user code" parameter as the "synchonous" interface procedures do not : indeed, the latter ones send only one request message and wait for the associated response message : there is no need of a "user code" in order to associate both messages.

Control_of_the_operation_of_an_actor (see sections 2.2, 2.3.2 and 4.6)

Prepare (Source port, Destination Port, Text of the message)

Note : this procedure pushes a message on the stack.

Select (Sending port, Local port)

Switch (Local port, Entry-point)

Time_Out (Local port, Sending port, Receiving port, Delay)

Return

Note : this procedure means the end of a processing-step; control is returned to the kernel along with the stack of messages.

External_Procedure_Call_Synchronisation (see section 4.1)

Ep_Call (Source port, Destination port, Text of the message, Delay)

Manipulation_of_actors (see section 2.4)

Create_Actor (Local port, User Code, Model of actor, Site of creation)

Note : this is the asynchronous interface procedure for the creation of an actor.

Synchronous_Create_Actor (Model of actor, Site of creation, Initial message, VAR Name of the created actor, VAR Name of the umbilical port of the created actor, VAR Diagnosis)

Note : this is the synchronous interface procedure for the creation of an actor; as respect to the previous procedure, three result parameters did appear : the creator actor must send the "initial message" onto the "umbilical port of the created actor".

Delete_Actor (Local port, User code, Name of the actor to be deleted)

Synchronous_Delete_Actor (Name of the actor to be deleted, VAR Diagnosis)

Delete_Myself (Local port)

Note : this procedure is asynchronous! In order to be sure not to be awaken to process a new message, the actor must issue for instance a Select (local port, local port).

Manipulation_of_ports (see section 2.3)

Create_Port (Local port, User code, Model of port, Name of the port to be created, Initial parameters)

Note : "Name of the port to be created" is an optional parameter : if present, the created port will be given that name (provided it is unique); if not present, the created port will be given a new global unique name.

Synchronous_Create_Port (Model of port, VAR Name of the port to be created, Initial parameters, VAR Diagnosis)

Open_Port (Local port, User code, Name of the port to be opened, Priority of the port)

Synchronous_Open_Port (Name of the port to be opened, Priority of the port, VAR Diagnosis)

Close_Port (Local port, User code, Name of the port to be closed)

Synchronous_Close_Port (Name of the port to be closed, VAR Diagnosis)

Delete_Port (Local port, User code, Name of the port to be deleted)

Synchronous_Delete_Port (Name of the port to be deleted, VAR Diagnosis)

Manipulation_of_interrupts (see section 3.2)

Interrupt_Associate (Local port, User code, Interrupt)

Note : the "Local port" is the port associated with "Interrupt".

Synchronous_Interrupt_Associate (Local port, Interrupt, VAR Diagnosis)

Interrupt_Dissociate (Local port, User code, Interrupt)

Synchronous_Interrupt_Dissociate (Local port, Interrupt, VAR Diagnosis)

Bibliography

[Banino 80] J.S. Banino, A. Caristan, M. Guillemont, G. Morisset, H. Zimmermann
CHORUS: an architecture for distributed systems
Rapport INRIA 42, (Novembre 1980), pp. 68

[Banino 82] J.S. Banino, J.C. Fabre
Distributed Coupled Actors: a CHORUS Proposal for Reliability
3rd International Conference on Distributed Computing Systems, Ft Lauderdale, Miami, Florida, U.S.A., (October 1982), pp. 7

[Fabre 82] J.C. Fabre
Un mécanisme de tolérance aux pannes sur l'architecture CHORUS
Thèse de Docteur Ingénieur, Toulouse, (Octobre 1982), pp. 200

[Finger 82] U. Finger, G. Médigue
Architectures multiprocesseurs: l'exemple de la SM90
Minis et Micros no 173, pp. 65, 69

[Grangé 82] J.L. Grangé
A mass transport service on high transmission rate satellite circuits - Some design considerations
(February 1982), pp. 16

[Guillemont 82a] M. Guillemont
The CHORUS distributed operating system: design and implementation
International Symposium on Local Computer Networks, Florence, Italy, (April 1982), pp. 207, 223

[Guillemont 82b] M. Guillemont
Intégration d'un système réparti, CHORUS, dans un langage de haut niveau, Pascal
Thèse de Docteur Ingénieur, Grenoble, (Mars 1982), pp. 250

[Guillemont 84] M. Guillemont
Comparative study of some distributed systems
TSI, vol 3, 1 (January 1984), pp. 15, 17

[ISO 82] ISO/TC97/SC16
Open Systems Interconnection - Transport Service Definition
DP 8072, (1982)

[Naffah 80] N. Naffah, B. Scheurer et AL
Description fonctionnelle du réseau expérimental DANUBE
(Juillet 1980), pp. 30

[Senay 83] C. Senay
Un système de désignation et de gestion de portes pour l'architecture répartie CHORUS
Thèse de Docteur Ingénieur, CNAM, (Décembre 83), pp. 200

[Softech 80] SOFTECH microsystems
UCSD PASCAL (TM), version II.0, User's manual
Softech microsystems publication, (February 1980), pp. 400

[Zimmermann 81] H. Zimmermann, J.S. Banino, A. Caristan, M. Guillemont, G. Morisset
Basic concepts for the support of distributed systems: the CHORUS approach
2nd International Conference on Distributed Computing Systems, Versailles, France, (April 1981), pp. 60, 66